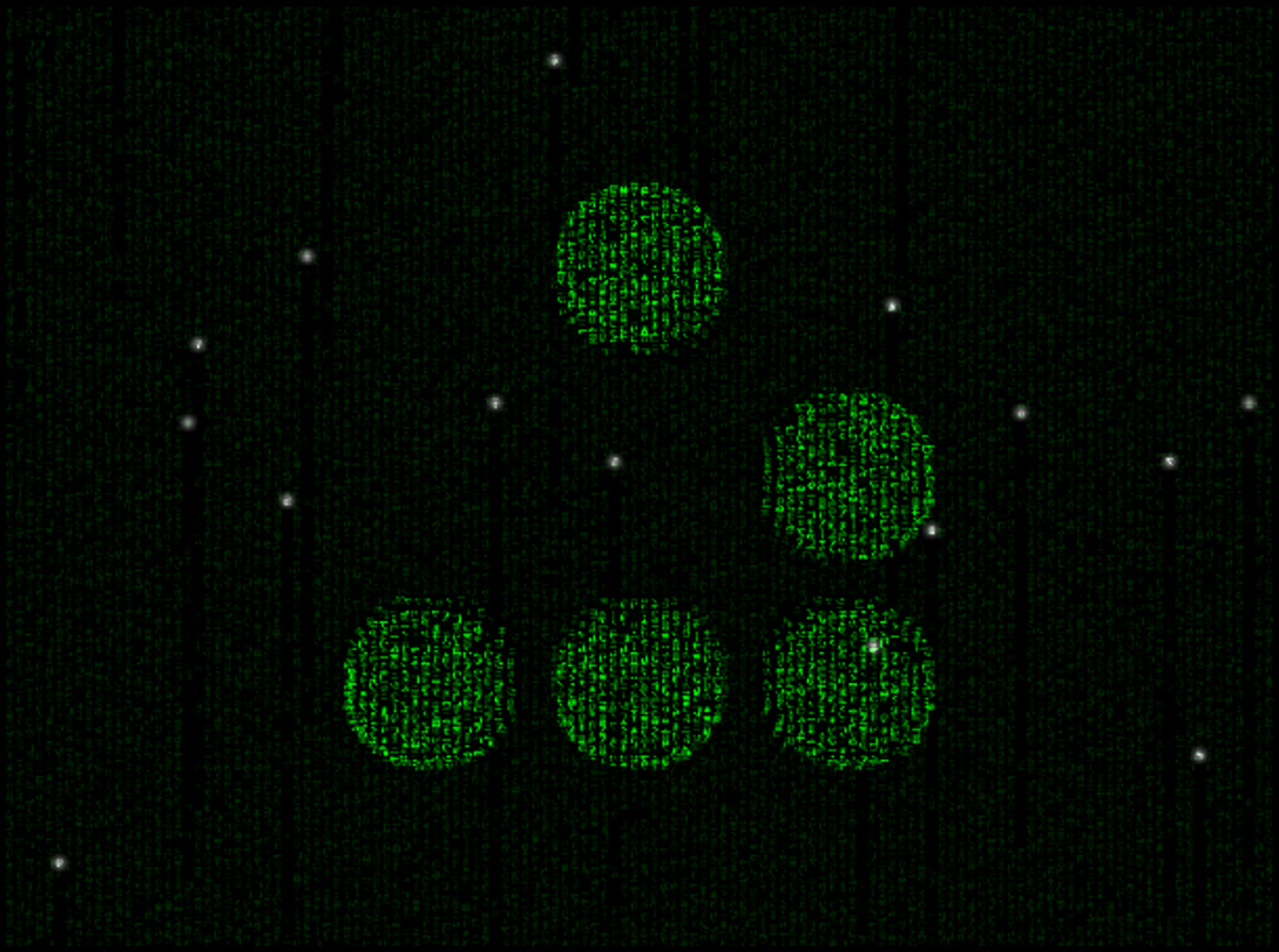


Боты для компьютерных игр

Илья Шпигорь

Боты для компьютерных игр

Илья Шпигорь

Эта книга продается по адресу http://leanpub.com/video-game-bots

Эта версия была опубликована 16.02.2021

* * * * *

Это книга с Leanpub book. Leanpub позволяет авторам и издателям участвовать в так называемом Lean Publishing - процессе, при котором электронная книга становится доступна читателям ещё до её завершения. Это помогает собрать отзывы и пожелания для скорейшего улучшения книги. Мы призываем авторов публиковать свои работы как можно раньше и чаще, постепенно улучшая качество и объём материала. Тем более, что с нашими удобными инструментами этот процесс превращается в удовольствие.

* * * * *

© 2019 - 2021 Илья Шпигорь

Оглавление

- Об авторе
- Благодарности
- Предисловие
- Классификация ботов
  - Задачи ботов
  - Игровое приложение
  - Виды ботов
  - Выводы
- Кликеры
  - Инструменты для разработки
  - Внедрение данных на уровне ОС
  - Перехват устройств вывода
  - Пример кликера для Lineage 2
  - Методы защиты от кликеров
- Внутриигровые боты
  - Инструменты для разработки
  - Организация памяти процесса
  - Доступ к памяти процесса
  - Пример бота для Diablo 2
  - Методы защиты от внутриигровых ботов
- Внеигровые боты
  - Инструменты для разработки
  - Сетевые протоколы
  - Перехват трафика
  - Пример бота для NetChess
  - Методы защиты от внеигровых ботов
- Специальные техники
  - Эмуляция устройств ввода

Перехват данных на уровне ОС

Заключение

Об авторе

Илья Шпигорь — разработчик программного обеспечения и поклонник открытого ПО. Имеет большой опыт работы со встраиваемыми системами и информационной безопасностью.

Сейчас работает над системами безопасности Ethernet сетей для автомобилей. До этого занимался системами обнаружения вторжений, авиасимуляторами для профессиональных пилотов и системами управления кораблями. Также участвовал в разработке эмулятора Wine и дистрибутива ALT Linux.

Илья интересуется автоматизацией процессов и исследованием возможностей новых языков программирования. В свободное время исследует уязвимости ПО и современные технологии ИИ.

Благодарности

Хочу поблагодарить всех, кто принимал участие в написании этой книги. Прежде всего спасибо Светлане Залогиной, которая первая увидела черновики и помогала мне с корректировкой стиля. Без неё эта книга никогда бы не была написана.

Спасибо Данилу Богданову и Эмилю Шайхилисламову, которые указали мне на технические ошибки и предложили несколько отличных идей.

Спасибо Руслану Пясецкому за то, что посвятил меня в тонкости криптографических алгоритмов.

Спасибо Алексею Пыльцыну за помощь с корректурой перевода книги на русский язык и переносом её на платформу Leanpub.

Также спасибо моей маме Елене Шпигорь, которая поддерживала меня на протяжении всей работы над книгой и помогала с корректурой.

Предисловие

Однажды, играя в любимую компьютерную игру, вы обнаруживаете, что без конца повторяете одни и те же действия. Возможно, этот процесс напомнит вам работу на старом ручном станке. Вы должны установить заготовку в зажим. Затем периодически жать ногой на педаль, чтобы сверло вращалось. Потянув рукоятку, вы направляете его на заготовку. Снова и снова вы повторяете эти действия для изготовления каждой детали. Но постойте. Мы живем в XXI веке, и человечество научилось автоматизировать простые, рутинные действия несколько десятилетий назад. Примерно такие мысли возникли у меня, когда я играл в компьютерную игру.

Я решил поискать возможности автоматизировать игровой процесс. С этой целью было просмотрено множество форумов и сайтов. К сожалению, большинство приложений, которые я нашел, содержало вредоносный код. Были программы без вирусов, но они отказывались работать как надо. В процессе моих поисков встретилось несколько подозрительных личностей со странными никнеймами, которые предлагали купить у них приложения способные (по их словам) решить все мои проблемы. Но мне показалось опрометчивым приобретать что-то без каких либо гарантий. Намного позже я понял, почему эти люди скрывали свои имена. В конце концов эти поиски не увенчались успехом.

Следующим моим шагом стала попытка написать программу автоматизации (называемую бот) самому. К сожалению, я столкнулся с серьезной нехваткой информации о подходах к решению этой задачи. Это показалось мне странным, учитывая что боты часто применяют сложные алгоритмы и используют методы из различных областей информационных технологий. Кроме того, разработка ботов имеет длинную историю и возникла отнюдь не вчера. Энтузиасты-одиночки и профессиональные программисты исследовали и применили множество решений для эффективной автоматизации игрового

процесса. Почему же никто из них не горит желанием поделиться своим опытом?

Эта книга – моя попытка исправить существующее положение вещей. В ней вы найдете полную классификацию ботов, которую я составил по результатам своих исследований. Мы подробно рассмотрим внутреннюю организацию различных типов ботов и напишем их несложные прототипы. Вы узнаете об эффективных инструментах для разработки, а также о существующих системах защиты (античит), способных обнаружить ботов.

Эта книга будет интересна всем игрокам, желающим по-новому взглянуть на игровой процесс. Она пригодится и тем, кто не интересуются разработкой программ, а хочет просто купить себе бота и использовать его. В этом им поможет обзорная информация о ботах и приемах их применения. Надеюсь, каждый найдет в этой книге что-то интересное и новое для себя.

Классификация ботов

В этой главе мы познакомимся с основными принципами работы ботов, исследуем историю их возникновения и эволюцию решаемых ими задач. Затем изучим архитектуру современных онлайн-игр. После этого мы рассмотрим две классификации ботов: по способу взаимодействия с игровым приложением и по методу перехвата и внедрения данных в него. Материалы этой главы помогут систематизировать наше дальнейшее изучение.

Задачи ботов

Для чего нужны боты? Наверняка, именно этот вопрос вы зададите, услышав о них впервые. Чтобы ответить на него, нам придётся обратиться к истории.

Одно из самых ранних упоминаний термина **игровой бот** встречается применительно к играм в жанре **шутер от первого лица** (first-person Shooter или FPS). Проблема появилась с момента возникновения режима **игрок-против-игрока** (player versus player или PvP), в котором пользователи могли соревноваться друг с другом. Некоторым из них хотелось подготовиться к состязанию самостоятельно, либо у них не было возможности подключаться к сети регулярно. Для этой цели требовались оппоненты, управляемые компьютерной программой, а не человеком.

Важно отметить, что новый соревновательный режим значительно отличается от однопользовательского, который возник намного раньше. В одиночной игре пользователь проходит уровни один за другим, решая головоломки и сражаясь с противниками. Поведение этих противников крайне примитивно: они стоят в предопределённых точках уровня и реагируют только при приближении игрока. Эти задачи решаются **игровым искусственным интеллектом**.

В соревновательном режиме жанра FPS от ИИ требуется большего. Он должен свободно перемещаться по игровому уровню, собирать оружие и боеприпасы, выбирать подходящий момент для нападения на противника и отступления. Другими словами ИИ должен вести себя или, по крайней мере, притворяться игроком-человеком. Именно такой вид ИИ получил название **бот**.

С развитием компьютерных игр возникли новые задачи для ИИ. Распространение интернета привело к росту популярности **массовых многопользовательских ролевых онлайн-игр** (massively multiplayer online role-playing game или MMORPG). Этот новый жанр имеет много общего с классическими **ролевыми играми** (role-playing game или RPG), но в отличие от них игровой процесс стал более растянутым по времени из-за большого числа участников. Кроме того, в MMORPG разработчики стремятся поддержать интерес пользователей как можно дольше. Эти особенности привели к более длительному развитию игрового персонажа. Теперь требуются недели, а иногда и месяцы, для выполнения квестов и добычи ресурсов. Благодаря этому повышается уровень персонажа, который важен для сражения с другими игроками. Именно этот соревновательный режим и является главной привлекательной чертой MMORPG.

Некоторым игрокам процесс развития персонажа может показаться скучным из-за постоянного повторения одних и тех же действий. Рано или поздно, они задумаются о способах его автоматизации. Разработчики некоторых MMORPG предоставляют средства для создания расширений, которые добавляют некоторую автономность персонажу. Но, как правило, таких средств нет или они оказываются недостаточными. Для расширения функциональности игры требуются возможности не предусмотренные разработчиками. Обычно такие расширения запрещены и блокируются программным путём, потому что издатель игры теряет из-за них деньги: благодаря автономности персонажа, игроки проводят меньше времени онлайн и совершают меньше внутриигровых покупок. Такие средства автоматизации в MMORPG были также названы ботами. Возможно, причина в том, что эти программы имитируют поведение игрока-человека точно так же, как в шутерах.

Автоматизация игрового процесса – не единственная задача, возникшая после появления новых жанров онлайн-игр. Некоторые увлечённые соперничеством пользователи начали искать пути обхода правил, чтобы получить нечестное преимущество. Например, приобрести необходимые ресурсы или дополнительную информацию о состоянии игры, изменить характеристики персонажей и т.д. Приложения для расширения функциональности игры с целью обхода её правил называются **читы** (cheats), **хаки** (hacks) и иногда боты. Это вызывает определённую путаницу. Жульничество в компьютерных играх – это не то же самое, что автоматизация. В этой книге мы проведём чёткую грань между читами и ботами. Боты – это средства для имитации поведения игрока, и именно их мы будем рассматривать.

Боты могут решать различные задачи. Они дают возможность пользователям тренироваться перед соревнованиями с другими людьми в шутерах и иных киберспортивных дисциплинах. Боты могут ускорять развитие персонажа в MMORPG. Они также дают конкурентное преимущество в соревновательных играх, путём модификации игрового процесса.

Игровое приложение

Перед изучением внутреннего устройства ботов, рассмотрим типичное игровое приложения. Принцип его работы не зависит от игрового жанра.

Сначала мы рассмотрим **онлайн-игру**. Иллюстрация 1-1 демонстрирует её логические элементы.

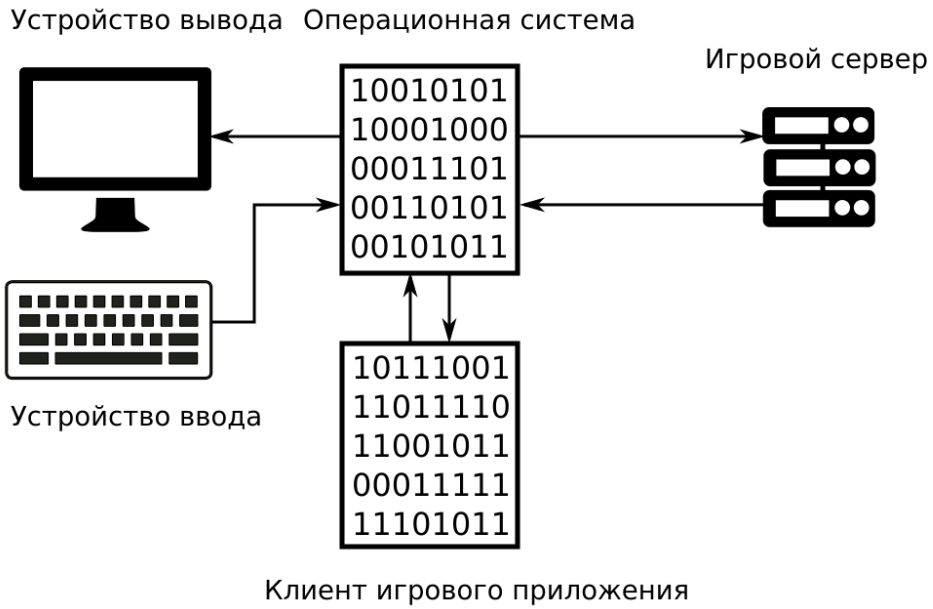


Иллюстрация 1-1. Элементы типичного приложения онлайн-игры

Запуская игру на компьютере, вы создаёте новый **вычислительный процесс** (process). Он имеет собственную **область памяти** (memory region), которая выделяется **операционной системой** (ОС). Память – только один из ресурсов предоставляемых ОС. Другими ресурсами являются устройства ввода-вывода: монитор, клавиатура, мышь, сетевая плата и т.д. Процессорное время тоже относится к одному из ресурсов. Оно определяет, как часто конкретный процесс получает управление в многозадачной ОС.

Возможно, вы спросите: “Зачем нужна ОС для запуска игры? Разве не проще работать приложениям вообще без неё?” ОС – это прежде всего удобная платформа для разработки. Без неё каждой компании, создающей программы, пришлось бы изобретать собственные средства для работы с устройствами ввода-вывода. Это потребовало бы много времени и усилий. Намного проще использовать **драйвера устройств** и **системные библиотеки**, предоставляемые ОС. Кроме того ни о какой **многозадачности** не могло бы идти и речи: всё процессорное время использовалось бы только одним приложением, как это было в **MS-DOS**.

Вернёмся к иллюстрации 1-1. Прямоугольники соответствуют элементам приложения, а стрелки – направлению передачи данных.

ОС обрабатывает команды работающего процесса для отображения картинки на мониторе или для отправки пакетов на сервер через сетевую плату. Также ОС уведомляет процесс о событиях на устройствах ввода. Например, нажатие клавиши на клавиатуре или получение пакета от сервера. ОС выполняет эти задачи с помощью драйверов устройств и системных библиотек. На иллюстрации они объединены в единый блок “Операционная Система”.

Рассмотрим обработку однократного действия игрока. В ней участвует несколько элементов, изображённых на иллюстрации. Предположим, что мы перемещаем персонажа. Для этого нажимаем клавишу на клавиатуре. Обработка нажатия состоит из следующих шагов:

1. **Устройство ввода -> Операционная система**
Клавиатура посылает сигнал о нажатии клавиши **контроллеру прерываний**. Это устройство передаёт сигналы в процессор в порядке очереди и с учётом приоритетов. На программном уровне они обрабатываются драйвером ОС.

2. **Операционная система -> Клиент игрового приложения**
ОС получает от драйвера событие, соответствующее нажатию клавиши. Затем ОС передаёт его дальше: процессу игрового приложения. Обычно, событие нажатия клавиши получает процесс, окно которого является активным в данный момент. Предположим, что активно игровое приложение.

3. **Клиент игрового приложения**
После получения события нажатия клавиши, процесс обновляет состояние игровых объектов в своей памяти. В нашем случае изменение касается местоположения персонажа.

4. **Клиент игрового приложения -> Операционная система**
Процесс должен сообщить игровому серверу о новом местоположении персонажа. Для этого надо отправить на сервер сетевой пакет с новой информацией. Процесс обращается к ОС через системную библиотеку.

Эта библиотека получает доступ к драйверу сетевой платы, который и отправляет пакет.

5. **Операционная система -> Игровой сервер**
Игровой сервер получает сетевой пакет. Затем он проверяет, соответствует ли новая информация о персонаже игровым правилам. Если проверка прошла успешно, сервер принимает эти данные и отправляет клиенту подтверждение. Если к серверу подключено несколько клиентов, он рассылает новую информацию о персонаже им всем.

6. **Операционная система -> Клиент игрового приложения**
Через контроллер прерываний сетевая плата посылает сигнал в процессор о получении сетевого пакета от игрового сервера. Сигнал обрабатывается драйвером. На уровне ОС создаётся соответствующее событие, которое передаётся процессу игрового приложения.

7. **Клиент игрового приложения**
Процесс извлекает из сетевого пакета код подтверждения игрового сервера. Если код сообщает об ошибке, местоположение персонажа остаётся неизменным. В противном случае процесс пометит в своей памяти, что новая информация о персонаже была успешно принята сервером.

8. **Клиент игрового приложения -> Операционная система**
Процесс игрового приложения обращается к системной библиотеке ОС (в случае Windows это обычно DirectX) для отображения на мониторе нового положения персонажа.

9. **Операционная система -> Устройство вывода**
Библиотека ОС выполняет необходимые расчёты и обращается к драйверу видеокарты для отрисовки картинки на экране.

Практически все действия игрока выполняются по описанному алгоритму независимо от устройства ввода (клавиатура, джойстик или мышь). В случае, если не требуется подтверждение действия со стороны игрового сервера (например, при открытии меню), алгоритм будет несколько отличаться.

Состояние игровых объектов может меняться как из-за действий игрока, так и из-за событий на стороне сервера (например, срабатывание таймера). Эти события будут обрабатываться по алгоритму, который состоит из рассмотренных выше шагов с шестого по девятый. В этом случае сервер уведомляет клиента об изменении. После этого процесс игрового приложения обновляет состояние объектов и перерисовывает картинку на экране.

Большинство современных онлайн-игр работают по рассмотренной нами схеме. Эта схема работы называется **архитектурой клиент-сервер**.

Иллюстрация 1-2 демонстрирует схему **однопользовательской** **PC-игры**. В отличие от онлайн-игры, здесь отсутствует сервер. Действия пользователя отражаются только на памяти процесса игрового приложения, в которой хранится состояние всех объектов.

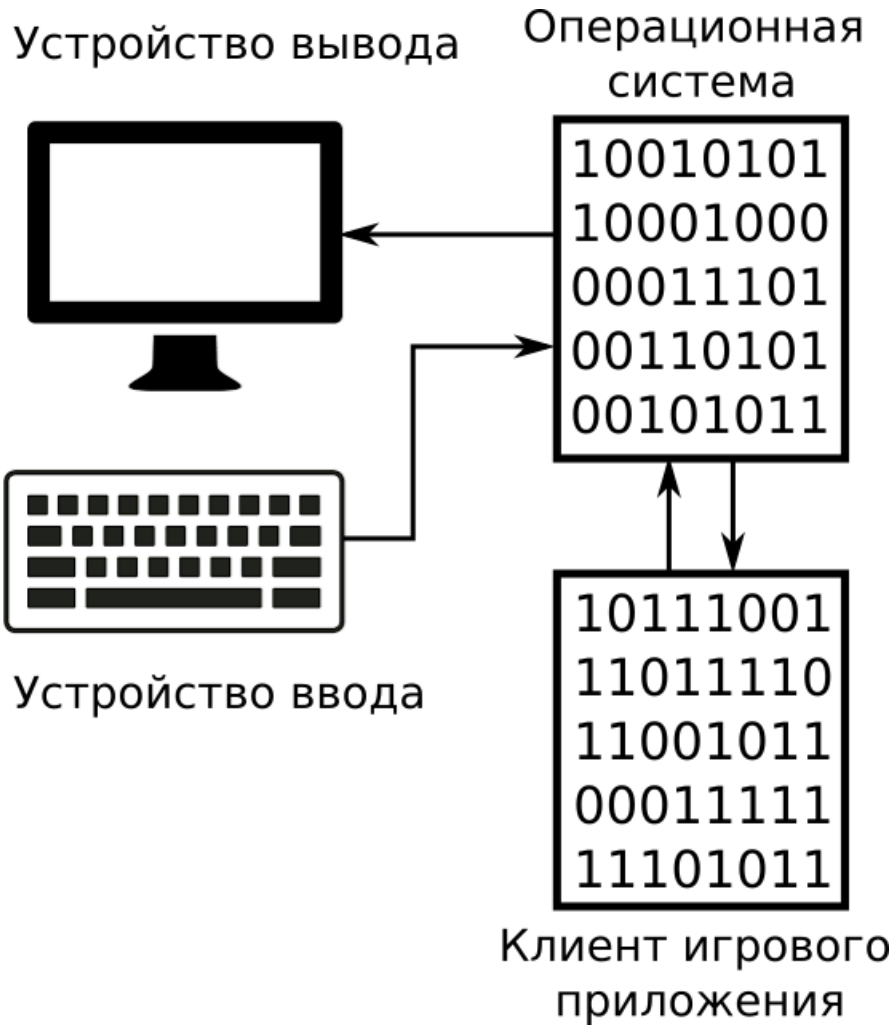


Иллюстрация 1-2. Элементы типичного приложения однопользовательской PC-игры

PC и онлайн-игры взаимодействуют с ресурсами ОС через системные библиотеки по одинаковым алгоритмам.

В случае онлайн-игры, состояние игровых объектов хранится на сервере и клиенте. При этом информация на стороне сервера более приоритетна. Это значит, что если информация клиента отличается, она будет заменена на ту, что хранится на сервере. Таким образом сервер контролирует корректность состояния игровых объектов. В случае однопользовательской PC-игры такого контроля нет.

Виды ботов

Попробуем классифицировать игровых ботов. Сразу возникает вопрос о том, по какому признаку следует относить бота к тому или иному виду. Единственного верного ответа здесь нет. Предлагаю рассмотреть ботов с двух точек зрения: их разработчиков и пользователей. Результат классификации в этих случаях получится разный.

Классификация сообщества игроков

Изучая информацию о ботах в Интернете, вы наверняка встретите термины **внутриигровой** (in-game) и **внеигровой** (out-game). Они широко используются и означают виды ботов, которые хорошо знакомы сообществу игроков. Основа для такой классификации – это способ взаимодействия с игровым приложением.

Внутриигровые боты получили своё название из-за того, что интегрируются в игровое приложение. Иллюстрация 1-3 демонстрирует такое взаимодействие. Специальные приёмы позволяют одному процессу ОС получить доступ к памяти другого процесса, либо загрузить в него произвольный исполняемый код. Таким образом бот манипулирует состоянием игровых объектов (например читает его, модифицирует и записывает обратно).

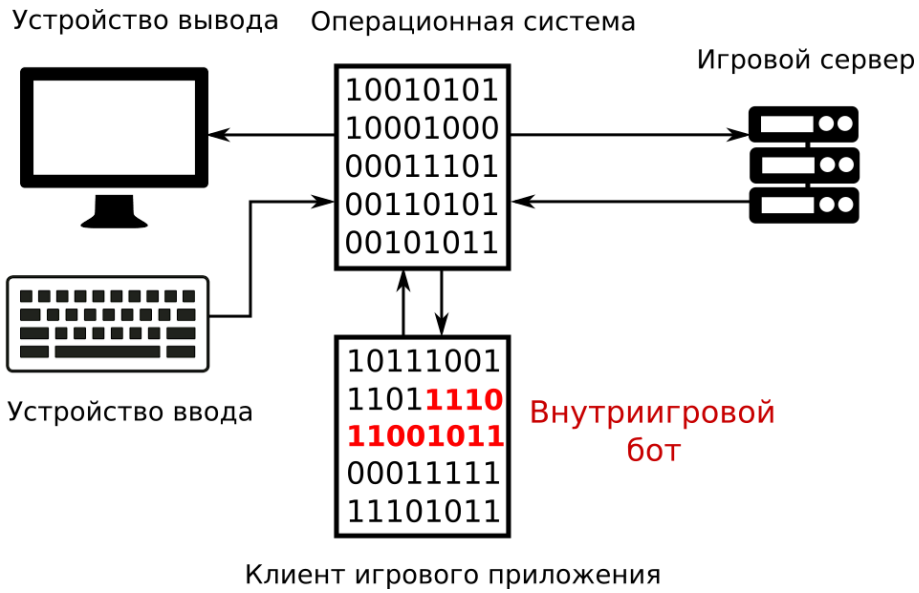


Иллюстрация 1-3. Схема работы внутриигрового бота

Внеигровые боты работают отдельно от игрового приложения, как на иллюстрации 1-4. Вместо чтения данных из памяти другого процесса, они используют возможности ОС для взаимодействия между процессами или сетевыми **хостами**. Хост – это компьютер подключённый к сети (например, клиент игрового приложения или сервер).

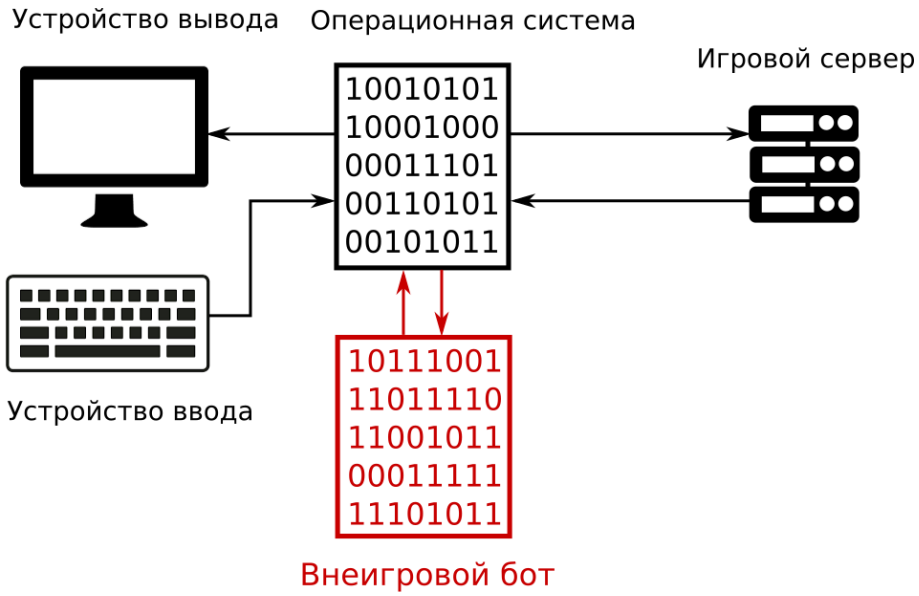


Иллюстрация 1-4. Внеигровой бот, работающий вместо игрового приложения

Существует два типа внеигровых ботов. Первый тип полностью подменяет собой игровое приложение. Вместо него вы запускаете бота, который взаимодействует напрямую с сервером. Самое сложное при таком подходе заключается в том, чтобы заставить сервер принять бота за настоящее игровое приложение.

Второй тип внеигровых ботов работает одновременно с игрой. В этом случае бот собирает информацию о состоянии игровых объектов и симулирует действия пользователя через системные библиотеки ОС. Иллюстрация 1-5 демонстрирует схему такой работы.

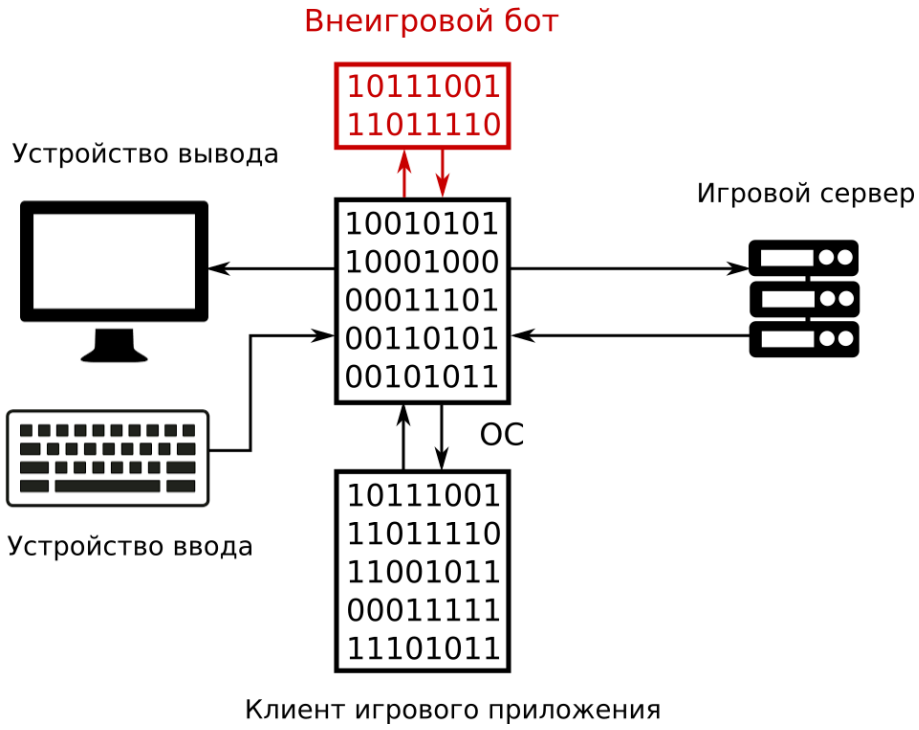


Иллюстрация 1-5. Внеигровой бот, работающий одновременно с игровым приложением

В Интернете также встречается упоминание кликеров. Их можно отнести ко второму типу внеигровых ботов. Особенность кликеров в том, что они симулируют нажатия клавиш и действия мыши через системные библиотеки ОС. При этом никакого доступа к памяти процесса игрового приложения или обмена сетевыми пакетами с сервером не происходит.

Классификация разработчиков

Классификация сообщества игроков была создана пользователями и полностью отвечает их нуждам. Познакомившись с ней, вы можете представить себе возможности и приёмы использования каждого вида ботов. Проблема в том, что эта классификация не отражает деталей реализации. Такая информация была бы полезна для разработчиков.

Чтобы построить классификацию удобную для разработчиков, попробуем взять за основу именно детали реализации ботов. Например, к разным видам будут относиться боты, читающие

состояние объектов из памяти игрового приложения, и те, которые обмениваются сообщениями с сервером.

Рассмотрим ещё раз схему приложения онлайн-игры. Отметим красными крестами точки, где бот может перехватить информацию о состоянии игровых объектов. Иллюстрация 1-6 демонстрирует результат.

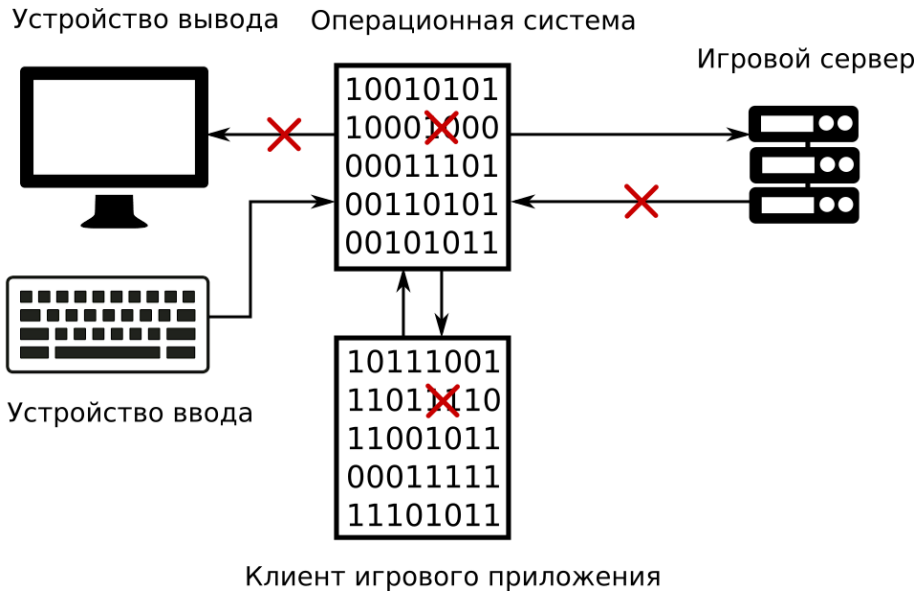


Иллюстрация 1-6. Точки перехвата информации ботом

Мы получили следующий список точек:

- **Устройство вывода**

С помощью системных библиотек ОС можно перехватывать данные, отправляемые на устройства вывода (например, монитор или звуковую карту). Предположим, что игровой объект отрисовывается на экране. Он имеет определённый цвет в зависимости от своего состояния. Бот может прочитать цвета пикселей отображённой на экране картинки и получить информацию об объекте.

- **Операционная система**

Бот может замещать или модифицировать системные библиотеки ОС или драйвера. Это позволит ему отслеживать взаимодействие игрового приложения с ОС. Альтернативное решение заключается в запуске игры в виртуальной машине или эмуляторе ОС (например, Wine). Как правило, эмуляторы имеют дополнительные средства журналирования событий. Эта информация позволит боту определить состояние игровых объектов.

- **Игровой сервер**

Сервер и клиент игрового приложения отправляют друг другу сетевые пакеты, каждый из которых содержит информацию об объектах или её часть. Перехватив достаточно пакетов, бот может сделать вывод о состоянии игры.

- **Клиент игрового приложения**

Бот может получить доступ к памяти процесса игрового приложения и прочитать из неё информацию. Системные библиотеки ОС предоставляют функции для этого.

Главная задача любого бота – это совершать игровые действия. При этом важно, чтобы он скрывал своё присутствие. То есть игровой сервер должен принимать действия бота так, как будто их совершил пользователь. Иллюстрация 1-7 демонстрирует точки на схеме, в которых бот может внедрять свои данные в приложение.

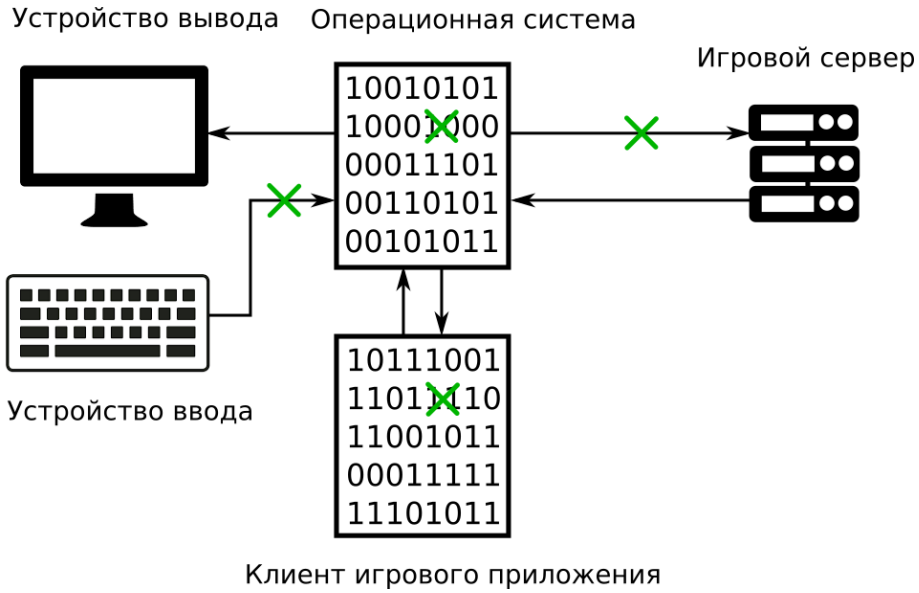


Иллюстрация 1-7. Точки внедрения ботом своих данных

Список точек получился следующий:

- **Устройство ввода**

Если бот контролирует устройство ввода, то с точки зрения ОС это достаточно сложно распознать. Разработчик может подменить стандартную клавиатуру или мышь устройством, которое получает команды от бота и симулирует нажатия клавиш.

- **Операционная система**

Так же как в случае с перехватом информации, бот может подменить компоненты ОС. Например, загрузить специальный драйвер, который уведомляет ОС о нажатии клавиши. При этом драйвер будет полностью под управлением бота. Кроме этого, есть системные библиотеки, которые предоставляют функции для встраивания событий нажатия клавиш в процесс игрового приложения.

- **Игровой сервер**

Бот может уведомлять сервер о своих действиях напрямую, посылая ему сетевые пакеты. Процедура их отправки может быть скопирована

у игрового приложения и перенесена в код бота.

- **Клиент игрового приложения**

Бот может встраивать свои действия и новые состояния объектов напрямую в память процесса игрового приложения. Таким образом сам игровой клиент будет обрабатывать эти действия и сообщать о них серверу.

В классификации разработчиков каждый бот может использовать одну из рассмотренных точек перехвата данных и внедрения своих действий. Таким образом мы получили 16 возможных комбинаций.

Сравнение ботов

Таблица 1-1 отображает соответствие между классификациями разработчиков и сообщества игроков. В столбцах указаны точки перехвата данных ботом, а в строках – точки внедрения действий. На пересечении полей и строк приведены названия из классификации игроков. Например, кликеры обычно перехватывают данные игры на уровне устройств вывода, а внедряют — на уровне ОС.

Таблица 1-1. Соответствие классификаций

	Перехват сетевых пакетов	**Чтение памяти**	**Перехват устройств вывода**	**Перехват на уровне ОС**
Внедрение в сетевые пакеты	Внеигровые боты	–	–	–
Внедрение в память	–	Внутриигровые боты	–	–
Внедрение на уровне устройств ввода	–	–	–	–
Внедрение на уровне ОС	–	–	Кликеры	–

Как видно из таблицы, классификация сообщества игроков покрывает только малую часть возможных вариантов реализаций ботов. Но эти варианты являются наиболее эффективными комбинациями точек перехвата и внедрения данных. Это не значит, что все три комбинации дают одинаковый результат. Каждая из них имеет свои достоинства и недостатки.

Перед тем как оценивать различные реализации ботов, определимся с критериями оценки. Рассмотрим три критерия:

1. Насколько трудозатратна реализация бота?
2. Насколько надёжен бот в смысле принятия верных решений?
3. Насколько сложно обнаружить бота системам защиты игры?

Кликеры наиболее просты для разработки и сопровождения. В то же время этот вид ботов наименее надёжен в использовании из-за большого количества совершаемых ошибок. Обнаружение кликеров – достаточно сложная задача для систем защиты игрового приложения.

Внеигровые боты наиболее трудоёмки для реализации. При этом их легко обнаружить. Их сильная сторона – максимальная надёжность в работе.

Внутриигровые боты являются средним вариантом между кликерами и внеигровыми ботами. Они сложнее в разработке чем первые, но проще чем вторые. Обнаружить их можно, но не так просто как внеигровых ботов. Надёжность работы выше чем у кликеров.

Почему результаты оценки ботов получились именно такими? Чтобы ответить на этот вопрос, рассмотрим каждый вариант реализации ботов с точки зрения оценки трудоёмкости разработки, надёжности и сложности обнаружения.

- **Сетевые пакеты**

Анализ сетевых пакетов является самым сложным методом перехвата данных. Разработчик бота должен реализовать протокол взаимодействия игрового клиента и сервера. Очевидно, документация

на этот протокол есть только у создателей игры. Обычно единственная доступная информация о протоколе – это перехваченные пакеты. Как правило, они зашифрованы и расшифровать их однозначно довольно сложно. С другой стороны, наиболее полная информация об игровых объектах может быть получена только напрямую от сервера. В этом случае игровой клиент ещё не успел её модифицировать или отфильтровать.

- **Память процесса игрового приложения**

Анализ памяти процесса – второй по сложности метод перехвата данных. Разработчики игр распространяют свои приложения в виде двоичных файлов. Эти файлы генерирует **компилятор** после прохода по исходному коду игры, который представляет собой читаемый текст. Проблема в том, что процесс компиляции необратим без неоднозначностей. Кроме того, системы защиты ещё более затрудняют изучение алгоритмов и структур данных игрового приложения. С другой стороны, анализ памяти процесса даёт почти такую же полную информацию о состоянии игровых объектов, как и анализ сетевых пакетов.
Внедрять действия бота в память процесса достаточно опасно, так как это может привести к завершению приложения с ошибкой.

- **Устройства вывода**

Перехват устройств вывода представляет собой одну из простейших техник сбора информации об игровых объектов. Но в то же время этот метод наименее надёжен. Например, алгоритмы распознавания изображений часто совершают ошибки, принимая один объект за другой. Эффективность этого подхода во многом зависит от интерфейса игры.

- **Устройства ввода**

Внедрение действия бота через эмулятор устройства ввода является эффективной техникой для обхода систем защиты игры. С другой стороны, необходимо купить это устройство и разработать прошивку

для него. Намного проще использовать внедрение действий бота на уровне ОС, если это допускает система защиты.

- **Операционная система**

Перехват данных на уровне ОС – это универсальный и надёжный метод. Существует несколько открытых проектов (например Direct3D 9 API Interceptor), которые позволяют подменять системные библиотеки. В этом случае игровое приложение взаимодействует с библиотеками, контролируемыми ботом. Они собирают информацию о вызываемых функциях ОС. Анализ этой информации позволит определить состояние игровых объектов.
Внедрение действий бота с помощью системных библиотек ОС достаточно просто реализовать. С другой стороны, система защиты легко обнаруживает эту технику.

В итоге мы можем заключить, что классификация сообщества игроков покрывает наиболее эффективные или простые для реализации комбинации техник перехвата и внедрения данных. В то же время она игнорирует неэффективные и редко встречающиеся комбинации. Мы будем следовать этой классификации на протяжении всей книги.

Выводы

В этой главе мы получили общее представление о ботах и их видах. Также мы рассмотрели некоторые аспекты их реализации. Теперь вы можете легко различить кликеров, внутриигровых и внеигровых ботов. Более того, вы представляете в общих чертах, как они работают, а также их сильные и слабые стороны.

Кликеры

Мы начнём изучение ботов с самого простого для реализации вида – кликеров. В начале этой главы мы рассмотрим широко используемые инструменты для разработки. Затем изучим техники встраивания данных в процесс игрового приложения на уровне ОС, а также перехвата устройства вывода. Чтобы закрепить полученные знания, мы напишем простого бота для игры Lineage 2. Этот небольшой проект поможет нам оценить достоинства и недостатки кликеров. В конце главы мы рассмотрим подходы для обнаружения этого вида ботов системами защиты.

Инструменты для разработки

Вы начинаете писать программу, чтобы решить какую-то проблему. При этом есть большая вероятность, что с подобной задачей кто-то уже сталкивался до вас. Скорее всего, для её решения уже были разработаны специальные инструменты. Поэтому лучшее, что вы можете сделать, перед тем как начать писать свой код, – это изучить существующие языки программирования, фреймворки и библиотеки. Если вам повезёт, вы найдёте несколько готовых решений, которые будет достаточно скомпоновать вместе для получения нужной функциональности. При этом важно не зацикливаться на использовании хорошо знакомых вам инструментов. Скорее всего, с их помощью вы сможете написать практически любое приложение, но на это уйдёт намного больше усилий, чем при использовании более подходящих средств.

В этом разделе мы рассмотрим несколько инструментов, которые хорошо подходят для разработки кликеров. Мы будем пользоваться ими для написания тестовых примеров. Но не исключено, что познакомившись с ними, вам в будущем удастся найти или купить более подходящие инструменты для своих проектов.

Язык программирования

AutoIt — один из самых популярных языков программирования для различных задач автоматизации приложений. У него много возможностей, которые ускоряют разработку:

- Простой для изучения синтаксис.
- Подробная доступная онлайн документация и поддержка сообщества на форумах.
- Хорошая интеграция с функциями ОС (**WinAPI**) и сторонними библиотеками.
- Встроенный редактор исходного кода.

AutoIt хорошо подходит для изучения программирования с нуля. Все примеры этой главы будут написаны на нём. В комментариях к ним мы рассмотрим WinAPI-функции, вызываемые через AutoIt. Таким образом, вам будет несложно переписать эти примеры на любом другом языке программирования.

AutoHotKey – это ещё один подходящий язык для написания кликеров. У него есть практически все возможности AutoIt. Основное различие этих языков в синтаксисе. Некоторые примеры этой главы будет проще и быстрее реализовать на AutoHotKey. Но этот язык немного более сложен в изучении.

Библиотеки обработки изображений

AutoIt имеет несколько встроенных средств обработки изображений. Но есть две библиотеки, которые значительно расширяют эти возможности.

Библиотека **ImageSearch** предоставляет функцию поиска указанного фрагмента изображения в окне игрового приложения. С её помощью бот может с высокой точностью и надёжностью определять месторасположение игровых объектов на экране.

Библиотека **FastFind** предоставляет продвинутые возможности поиска определённой комбинации пикселей в окне приложения. Например, поиск ближайшего к указанной точке пикселя заданного цвета. Это может быть полезно для обнаружения игровых объектов в случаях, когда библиотека ImageSearch не справляется (например, с 3D-графикой).

Инструменты анализа изображений

Для отладки кликеров могут понадобиться средства анализа изображений. Типичная задача отладки заключается в определении точных координат и цвета какого-то пикселя на скриншоте игры. Эта информация позволит проверить данные, поступающие на вход алгоритмов бота.

Существует множество подобных утилит, и вы легко найдёте их с помощью Google. Я предпочитаю приложение **ColorPix**, в котором есть все необходимое для решения наших задач.

Редакторы исходного кода

В дистрибутив языка AutoIt входит адаптированная версия редактора **SciTE**. Он хорошо подходит для написания и отладки AutoIt-скриптов. Если же вы планируете использовать другой язык (например, Python или AutoHotKey), вам понадобится более универсальный редактор. **Notepad++** будет подходящим решением для разработки небольших скриптов. Для C++ и C# лучше всего использовать **Visual Studio Community**.

Перехват API

В наших примерах мы будем писать скрипты на высокоуровневом языке программирования AutoIt. Это означает, что каждая инструкция, написанная на нём, скрывает несколько вызовов более низкоуровневых функций, предоставляемых ОС. Для того чтобы лучше понимать алгоритмы бота и исправлять ошибки в них, нам следует изучить внутреннюю работу функций AutoIt. Кроме того, эта информация позволит вам переписать примеры этой главы на другом языке программирования.

Существует несколько инструментов для перехвата вызова функций ОС. Я использовал бесплатное приложение **API Monitor v2**. У него есть следующие возможности:

- Фильтрация всех перехваченных вызовов.
- Сбор информации об анализируемом процессе.
- Декодирование входных и выходных параметров вызываемых функций.
- Просмотр памяти процесса.

Список всех возможностей приложения доступен на сайте разработчиков.

Внедрение данных на уровне ОС

Windows API

Главная задача любой ОС – это управление программными и аппаратными ресурсами компьютера, а также предоставление к ним доступа для запущенных процессов. Аппаратные ресурсы мы уже рассматривали. Это – память, процессорное время, периферийные устройства. К программным ресурсам относятся все приложения и компоненты ОС, установленные на компьютере. Примером этого типа ресурсов являются системные библиотеки Windows, предоставляющие алгоритмы для решения различных задач.

В этой книге мы рассматриваем только ОС Windows. На ней вы сможете запустить все приведённые примеры. В дальнейшем для простоты под ОС всегда будем подразумевать Windows.

Иллюстрация 2-1 демонстрирует интерфейс ОС, через который предоставляется доступ к её ресурсам. Каждый запущенный процесс может обратиться к Windows с запросом на выполнение какого-то действия (например создание нового окна, отправки сетевого пакета, выделения дополнительной памяти и т.д.). Для каждого из таких запросов у ОС есть соответствующая **функция** (или подпрограмма). Функции, которые решают задачи из одной области (например работа с сетью), собраны в отдельные системные библиотеки.

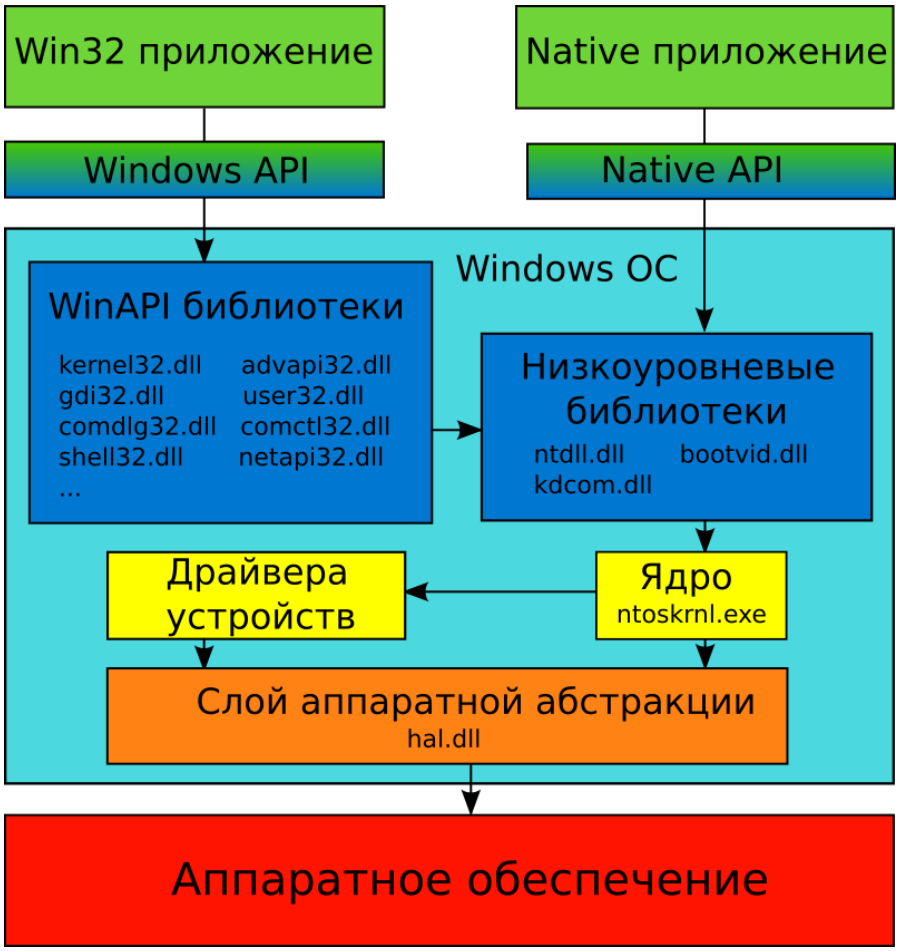


Иллюстрация 2-1. Доступ к ресурсам ОС через WinAPI

Способ, которым процесс может вызвать системную функцию, строго определён, хорошо задокументирован и остаётся неизменным для данной версии ОС. Такое взаимодействие можно сравнить с юридическим договором: если процесс выполняет предварительные условия для вызова функции, ОС гарантирует указанный в документации результат. Такой договор называется **интерфейс прикладного программирования Windows** (Windows API или WinAPI).

Программное обеспечение очень гибко и легко меняется согласно возникающим требованиям. Так каждое обновление Windows вносит изменения в некоторые детали реализации ОС (например, в какую-то системную библиотеку). Эти детали реализации связаны между собой: типичный случай – одна библиотека вызывает функции другой. Таким образом, даже небольшое изменение может оказать значительное влияние

на систему в целом. То же самое справедливо и для игрового приложения. Единственное, что позволяет программному обеспечению работать в этом море постоянных изменений – это надёжные интерфейсы. Именно WinAPI гарантирует согласованное состояние системы и обеспечивает совместимость между новыми версиями ОС и приложения.

На иллюстрации 2-1 приведены два типа приложений. **Win32-приложение** взаимодействует с подмножеством системных библиотек через WinAPI интерфейс. Win32 – это историческое название, которое возникло в первой 32-битной версии Windows (Windows NT). Библиотеки, доступные через WinAPI, также известные как **WinAPI библиотеки**, содержат функции, оперирующие сложными абстракциями: элемент управления, файл и т.д.

Второй тип приложений называется **нативные** (native, поэтому иногда переводится как родной). Они взаимодействуют с более низкоуровневыми библиотеками и ядром Windows через **Native API**. Преимущество этих библиотек в том, что они становятся доступны на раннем этапе загрузки системы, когда многие функции ОС ещё не работоспособны. Функции этих библиотек оперируют простыми абстракциями, такими как страница памяти, процесс, поток и т.д. Примеры нативных приложений: утилита для разбивки жёсткого диска, антивирус до старта ОС, программа восстановления Windows.

Библиотеки WinAPI вызывают функции низкоуровневых библиотек. Такой подход позволяет составлять сложные абстракции из более простых. Низкоуровневые библиотеки в свою очередь вызывают функции ядра.

Драйвера предоставляют упрощённое представление устройств для системных библиотек. Это представление включает в себя набор функций, которые выполняют характерные для данного устройства действия. WinAPI и низкоуровневые библиотеки обращаются к драйверам через функции ядра.

Слой аппаратной абстракции (Hardware Abstraction Layer или HAL) – это модуль ядра ОС, который предоставляет универсальный доступ к различному аппаратному обеспечению. HAL нужен, чтобы облегчить портирование и сопровождение Windows на новых аппаратных платформах. Функции HAL используются ядром ОС и драйверами устройств.

Симуляция нажатий клавиш

Теперь мы рассмотрим технику симуляции нажатий клавиш. Это наиболее простой метод контроля ботом игрового приложения.

Нажатия клавиш в активном окне

Рассмотрим, какие возможности предлагает AutoIt для решения нашей задачи. В списке доступных функций есть функция `Send`. Мы воспользуемся ею в тестовом скрипте, который будет нажимать клавишу “a” в окне приложения Notepad (Блокнот).

Алгоритм работы нашего скрипта выглядит следующим образом:

1. Найти окно Notepad среди всех открытых окон.
2. Переключится на него.
3. Симулировать нажатие клавиши “a”.

Для поиска окна приложения мы воспользуемся функцией `WinGetHandle`. Её первый параметр является обязательным и может быть как заголовком окна, так и его классом. Функция возвращает **дескриптор** (handle) окна. Дескриптор – это структура данных, которая представляет некоторый ресурс или объект ОС. Большинство функций WinAPI оперируют этими структурами при работе с объектами.

Указывать класс окна при вызове функции `WinGetHandle` предпочтительнее. Всегда есть вероятность, что окна некоторых работающий приложений будут иметь одинаковые заголовки (например пустые).

Для чтения класса окна Notepad необходимо выполнить следующие шаги:

1. Запустить приложение Au3Info. Вы можете найти его в каталоге установки AutoIt. Путь к приложению по умолчанию: `C:\Program Files (X86)\AutoIt3\Au3Info.exe`.
2. Перетащить иконку “Finder Tool” на окно Notepad и отпустить.

Вы увидите результат, приведённый на иллюстрации 2-2.

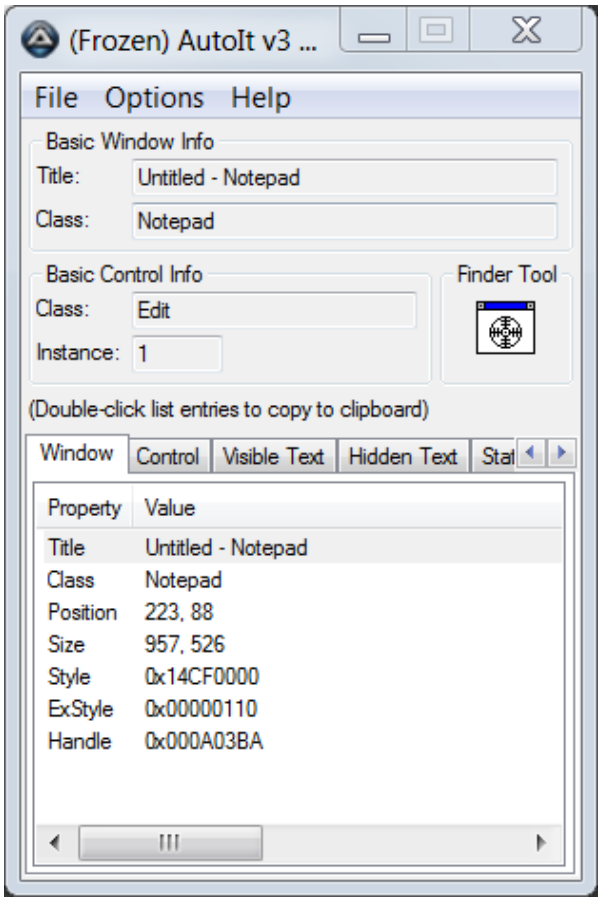


Иллюстрация 2-2. Приложение AutoIt3 Info

Класс окна Notepad отображается на панели “Basic Info Window”. Этот класс – “Notepad”.

Скрипт `Send.au3`, представленный в листинге 2-1, симулирует нажатие клавиши “a”.

Листинг 2-1. Скрипт `Send.au3`

```
1 $hWnd = WinGetHandle("[CLASS:Notepad]")
2 WinActivate($hWnd)
3 Send("a")
```

В первой строке мы получаем дескриптор окна Notepad с помощью функции `WinGetHandle`. Далее мы переключаем фокус ввода на это окно функцией `WinActivate`. Последнее действие – симуляция нажатия клавиши “a”.

Чтобы запустить этот скрипт, создайте в вашем редакторе исходного кода файл с именем `Send.au3` и скопируйте в него приведённый выше код. Запустите скрипт двойным щелчком по иконке этого файла.

Функция Send

Функция `Send` представляет собой обёртку над WinAPI вызовом. Мы можем выяснить что это за вызов с помощью приложения API Monitor, которое перехватит все обращения к WinAPI скрипта `Send.au3`.

Для подключения API Monitor к работающему процессу выполните следующие шаги:

1. Запустите 32-битную версию API Monitor.
2. Переключитесь на панель “API Filter” щелчком мыши. Нажмите комбинацию клавиш Ctrl+F, чтобы открыть диалог поиска. Введите в поле “Find what:” текст “Keyboard and Mouse Input” и нажмите кнопку “Find Next”. Закройте диалог поиска и активируйте найденный флажок (check box) “Keyboard and Mouse Input”.
3. Нажмите Ctrl+M для открытия диалога “Monitor New Process”. Выберите приложение `AutoIt3.exe` в поле “Process” и нажмите кнопку “OK”. По умолчанию путь к этому приложению должен быть следующий: `C:\Program Files (x86)\AutoIt3\AutoIt3.exe`.
4. В открывшемся диалоге “Run Script” выберите скрипт `Send.au3`. Сразу после этого начнётся его выполнение.
5. Переключитесь на панель “Summary” окна API Monitor. По нажатию Ctrl+F откройте диалог поиска и с его помощью найдите текст ‘a’ (с одинарными кавычками).

Иллюстрация 2-3 демонстрирует ожидаемый результат. Согласно перехваченным вызовам, `VkKeyScanW` – это единственная WinAPI-функция, получившая символ “a” в качестве параметра. Если мы обратимся к официальной документации WinAPI, выяснится что эта функция не выполняет нажатия клавиши. Она вместе с функцией `MapVirtualKeyW` только подготавливает параметры для вызова `SendInput`, который и симулирует нажатие.

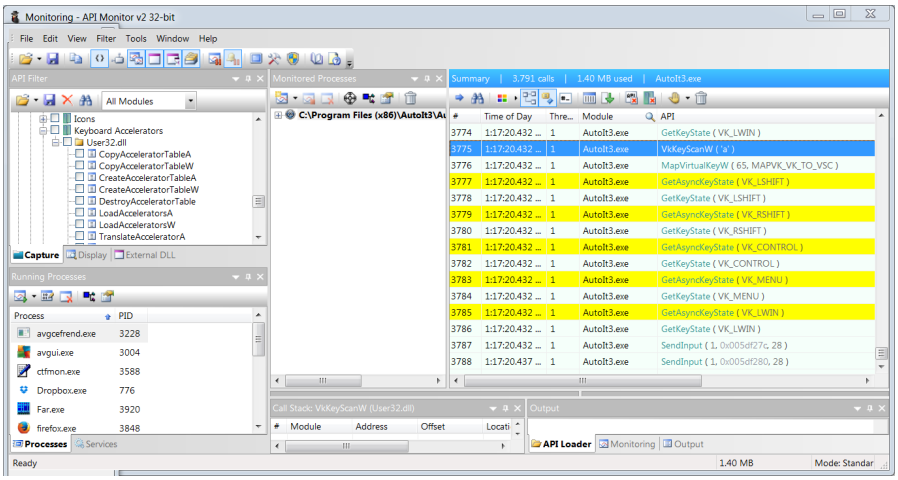


Иллюстрация 2-3. Перехват вызовов WinAPI с помощью API Monitor

Мы узнали достаточно, чтобы симулировать нажатие клавиши “a” напрямую через WinAPI вызовы. Удалим третью строчку скрипта `Send.au3` и заменим её новым блоком кода. При этом оставим первые два вызова `WinGetHandle` и `WinActivate` без изменений. Листинг 2-2 демонстрирует получившийся результат.

Листинг 2-2. Скрипт `SendInput.au3`

```
$hWnd = WinGetHandle("[CLASS:Notepad]")
WinActivate($hWnd)

Const $KEYEVENTF_UNICODE = 4
Const $INPUT_KEYBOARD = 1
Const $iInputSize = 28

Const $tagKEYBDINPUT = _
    'word wVk;' & _
    'word wScan;' & _
    'dword dwFlags;' & _
    'dword time;' & _
    'ulong_ptr dwExtraInfo'

Const $tagINPUT = _
    'dword type;' & _
    $tagKEYBDINPUT & _
    ';dword pad;'

$tINPUTs = DllStructCreate($tagINPUT)
$pINPUTs = DllStructGetPtr($tINPUTs)
$iINPUTs = 1
$Key = AscW('a')

DllStructSetData($tINPUTs, 1, $INPUT_KEYBOARD)
DllStructSetData($tINPUTs, 3, $Key)
DllStructSetData($tINPUTs, 4, $KEYEVENTF_UNICODE)

DllCall('user32.dll', 'uint', 'SendInput', _
        'uint', $iINPUTs, 'ptr', $pINPUTs, 'int', $iInputSize)
```

Здесь мы используем функцию AutoIt `DllCall`. С её помощью можно вызвать код **динамической библиотеки** (DLL), написанной на языке C или C++. В данном случае мы делаем WinAPI вызов `SendInput`. Его входные параметры должны иметь типы, согласно документации WinAPI. Некоторые из этих типов AutoIt не поддерживает на уровне синтаксиса. Поэтому нам нужны дополнительные шаги, чтобы подготовить эти параметры.

Таблица 2-1 демонстрирует входные параметры функции `DllCall`.

Таблица 2-1. Входные параметры функции `DllCall`

Параметр	Описание
`user32.dll`	Имя библиотеки, функцию которой требуется вызвать.
`uint`	Тип возвращаемого значения функции.
`SendInput`	Имя функции.
`uint`, `$iINPUTs`	Пары тип-переменная. Переменные
`ptr`, `$pINPUTs`	являются входными параметрами
`int`, `$iInputSize`	функции.

Согласно WinAPI документации, декларация функции `SendInput` выглядит следующим образом:

```
1 UINT SendInput(UINT cInputs, LPINPUT pInputs, int cbSize);
```

Строчку вызова функции `DllCall` на AutoIt можно представить эквивалентом на языке C++:

```
1 SendInput(iINPUTs, pINPUTs, iInputSize);
```

Рассмотрим входные параметры, переданные нами в `SendInput`:

1. `iINPUTs` – количество структур типа `INPUT`, которые передаются вторым параметром.
2. `pINPUTs` – указатель на массив структур типа `INPUT` из одного элемента. Этот массив подготавливается в несколько этапов. Сначала мы объявляем строки `KEYBDINPUT` и `INPUT` с описанием полей соответствующих структур. При этом `KEYBDINPUT` является вторым

полем `INPUT`. Такое отношение называется **вложенные структуры** (nested structure). На следующем шаге создаются структуры в формате языка C++ через вызов `DllStructCreate`. Результат сохраняется в переменной `tINPUTs`. С помощью функции `DllStructGetPtr` мы получаем указатель на эту структуру и помещаем его в `pINPUTs`. Запись значений полей C++ структуры происходит через вызов `DllStructSetData`. Обратите внимание, что вторым параметром в `DllStructSetData` передаётся номер поля, начиная с единицы. В случае вложенных структур их поля нумеруются последовательно. То есть элемент 1 соответствует полю `dword type` структуры `INPUT`, а элемент 3 – полю `word wScan` структуры `KEYBDINPUT`.
3. `iInputSize` – размер одной структуры `INPUT` в байтах. В нашем случае это константное значение, рассчитанное по формуле:

```
dword + (word + word + dword + dword + ulong_ptr) + dword =
4 + (2 + 2 + 4 + 4 + 8) + 4 = 28
```

Слагаемые в скобках – это размеры полей вложенной структуры `KEYBDINPUT`.

Может быть непонятно, откуда взялись последние четыре байта в приведённой выше формуле. Возможно, вы обратили внимание, что объявленная в скрипте `Send.au3` структура `INPUT` имеет последнее поле типа `dword` с именем `padding` (набивка). Оно не используется и служит для **выравнивания данных**. Рассмотрим это поле подробнее.

Определение структуры `INPUT` согласно документации WinAPI выглядит следующим образом:

```
typedef struct tagINPUT {
  DWORD type;
  union {
    MOUSEINPUT    mi;
    KEYBDINPUT    ki;
    HARDWAREINPUT hi;
  };
} INPUT, *PINPUT;
```

Вложенная структура `KEYBDINPUT` на самом деле помещена в блок `union` с другими структурами `MOUSEINPUT` и `HARDWAREINPUT`. Это означает, что под все три структуры будет выделена одна и та же область памяти. Но использоваться она будет только одной из них. Так как область одна, её размер должен соответствовать самой большой структуре, которой

является MOUSEINPUT. Она больше KEYBDINPUT на одно поле типа dword, т.е. на четыре байта. Именно из-за него мы добавили padding в наше определение KEYBDINPUT для выравнивания.

Скрипт SendInput.au3 демонстрирует преимущества высокоуровневых языков, таких как AutoIt. Они скрывают от пользователя множество несущественных деталей. Это позволяет оперировать простыми абстракциями и функциями. Кроме того, приложения, написанные на таких языках, короче и яснее.

Нажатия клавиш в неактивном окне

Функция AutoIt Send симулирует нажатия клавиш в активном окне. Другими словами, вы не можете свернуть это окно или переключится на другое, что в некоторых случаях неудобно. Функция ControlSend позволяет обойти такое ограничение. Мы можем переписать скрипт Send.au3 с использованием ControlSend, как демонстрирует листинг 2-3.

Листинг 2-3. Скрипт ControlSend.au3

```
1 $hWnd = WinGetHandle("[CLASS:Notepad]")
2 ControlSend($hWnd, "", "Edit1", "a")
```

Третьим параметром в функцию ControlSend передаётся **элемент интерфейса** (control), который получает симулируемое нажатие клавиши. Указать на него можно несколькими способами. В нашем случае, мы передаём класс элемента “Edit1”. Его можно узнать с помощью утилиты Au3Info точно так же, как и класс окна.

Применив API Monitor, мы узнаем, что ControlSend внутри себя вызывает WinAPI-функцию SetKeyboardState. В качестве упражнения предлагаю вам переписать скрипт ControlSend.au3 так, чтобы он вызывал SetKeyboardState напрямую.

Скрипт ControlSend.au3 работает корректно во всех случаях, кроме симуляции нажатия клавиши в развёрнутом на весь экран окне приложения DirectX. Проблема заключается в том, что такое окно не имеет элементов интерфейса. Чтобы её решить, достаточно просто не указывать третий параметр controlID функции ControlSend. Листинг 2-4 демонстрирует исправленный скрипт.

Листинг 2-4. Скрипт ControlSendDirectX.au3

```
1 $hWnd = WinGetHandle("Warcraft III")
2 ControlSend($hWnd, "", "", "a")
```

Этот скрипт ищет окно игры Warcraft 3 по его заголовку и симулирует в нём нажатие клавиши “a”. Узнать заголовок окна DirectX-приложения иногда бывает сложно, потому что не всегда можно выйти из полноэкранного режима. В этом случае утилиты вроде Au3Info вам не помогут. Вместо них с этой задачей справится API Monitor. Если в окне приложения вы наведёте курсор мыши на интересующий вас процесс на панели “Running Process”, вы увидите заголовок окна этого приложения, как показано на иллюстрации 2-4.

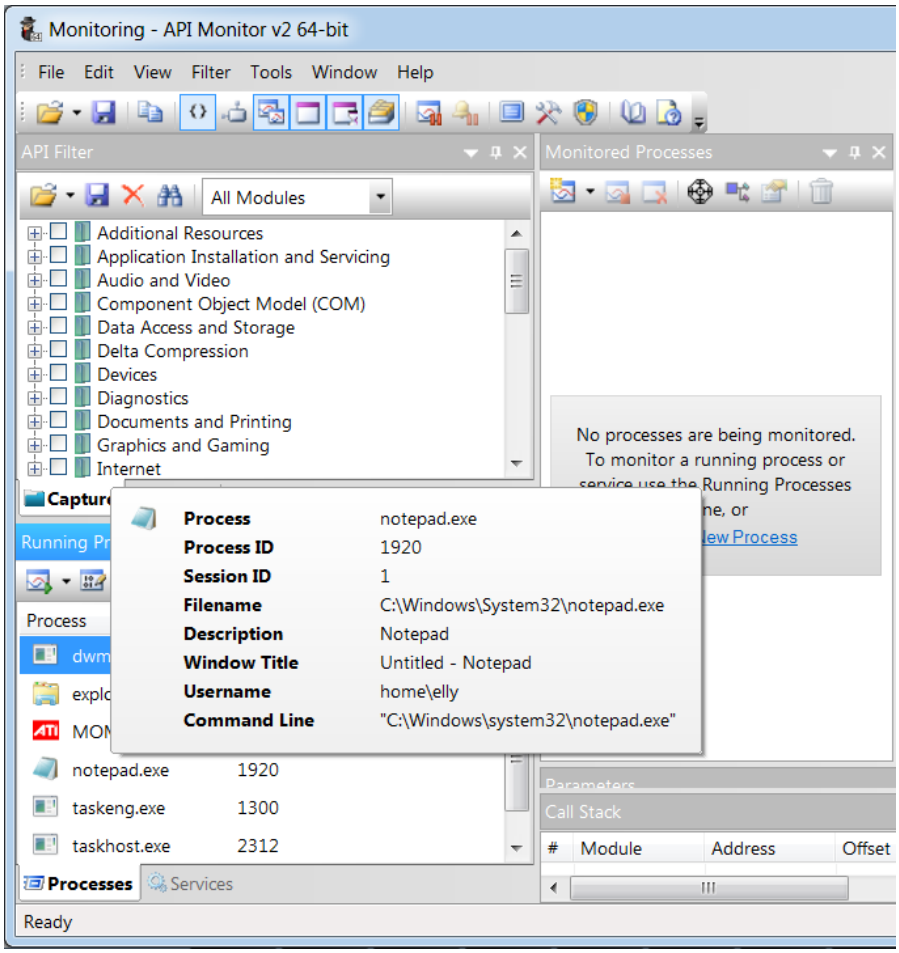


Иллюстрация 2-4. Чтение заголовка окна приложения в API Monitor

Если вы не можете найти нужный процесс на панели “Running Process”, попробуйте запустить API Monitor с правами администратора. Если это не помогло и у вас установлена 64-битная версия Windows, надо запустить

обе версии API Monitor – 32- и 64-битную. В одной из них процесс должен появиться.

Заголовок некоторых окон в полноэкранном режиме пустой. Из-за этого вы не сможете передать его в функцию `WinGetHandle` и получить дескриптор. Тогда альтернативным решением будет передавать класс окна. К сожалению, с помощью API Monitor эту информацию не удастся прочитать.

Чтобы получить класс окна, открытого в полноэкранном режиме, вы можете воспользоваться скриптом AutoIt, приведённом в листинге 2-5.

Листинг 2-5. Скрипт `GetWindowTitle.au3`

```
#include <WinAPI.au3>
Sleep(5 * 1000)
$handle = WinGetHandle('[Active]')
MsgBox(0, "", "Title   : " & WinGetTitle($handle) & @CRLF _
       & "Class : " & _WinAPI_GetClassName($handle))
```

После запуска скрипт ждёт пять секунд, в течение которых вы должны переключиться на интересующее вас окно. После этого его заголовок и класс будут выведены в открывшемся диалоговом окне.

Рассмотрим подробнее скрипт `GetWindowTitle.au3`. В первой строке стоит **ключевое слово** (keyword) `include`. С его помощью AutoIt включает содержание указанного скрипта `WinAPI.au3` в текущий. В `WinAPI.au3` реализована нужная нам функция `_WinAPI_GetClassName`. Она возвращает класс окна по его дескриптору. Далее с помощью функции `Sleep` скрипт ждёт пять секунд. После этого дескриптор активного в данный момент окна сохраняется в переменную `handle`. Функция `MsgBox` создаёт диалоговое окно, в котором выводится результат. Заголовок окна возвращает функция `WinGetTitle`.

Симуляция действий мыши

В некоторых играх для управления персонажем достаточно только клавиатуры. Однако в большинстве случаев игрок должен пользоваться и клавиатурой, и мышью. AutoIt предлагает несколько функций, которые позволят симулировать основные действия мыши: щелчки, перемещение курсора, зажимание кнопки.

Действия мыши в активном окне

Мы воспользуемся графическим редактором Microsoft Paint для тестирования скриптов, симулирующих действия мыши. Самое простое действие – это однократный щелчок в указанной точке экрана. Листинг 2-6 демонстрирует соответствующий скрипт.

Листинг 2-6. Скрипт `MouseClick.au3`

```
$hWnd = WinGetHandle("[CLASS:MSPaintApp]")
WinActivate($hWnd)
MouseClick("left", 250, 300)
```

Для тестирования этого скрипта выполните следующее:

1. Запустите приложение Paint.
2. Переключитесь на инструмент “Кисти” (Brushes).
3. Запустите скрипт `MouseClick.au3`.

Скрипт нарисует чёрную точку по координатам x = 250, y = 300. Их корректность вы можете проверить с помощью утилиты ColorPix.

Для симуляции щелчка мыши мы использовали функцию AutoIt `MouseClick`. Она принимает пять входных параметров, первые три из которых являются обязательными:

1. Кнопка мыши для щелчка. Основные варианты: левая (left), правая (right), средняя (middle).
2. Координата X позиции курсора.
3. Координата Y позиции курсора.
4. Число последовательных щелчков.
5. Скорость мыши для перемещения курсора в указанные координаты.

Внутри себя `MouseClick` вызывает WinAPI-функцию `mouse_event`.

Координаты позиции курсора можно задавать в одном из трёх режимов, представленных в таблице 2-2.

Таблица 2-2. Режимы координат, поддерживаемые WinAPI

Режим	Описание
0	Координаты относительно левой верхней точки активного окна.
1	Абсолютные координаты экрана. Это режим по умолчанию.
2	Координаты относительно левой верхней точки клиентской части окна (без заголовка, меню и границ).

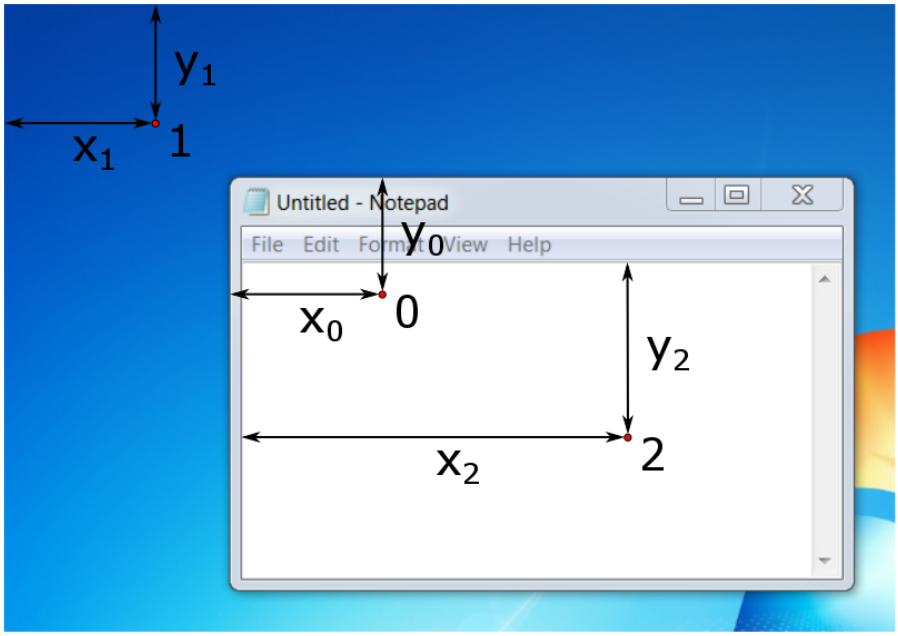


Иллюстрация 2-5. Режимы координат, поддерживаемые WinAPI

Рассмотрим иллюстрацию 2-5. Каждый номер соответствует режиму координат из таблицы 2-1. Например, точка с номером “0” демонстрирует режим относительно активного окна. Её координаты X0 и Y0.

Функция AutoIt `Opt`, вызванная с первым параметром `MouseCoordMode`, позволяет выбрать режим координат для текущего скрипта. Листинг 2-7 демонстрирует выбор координат относительно клиентской части окна в скрипте `MouseClick.au3`.

Листинг 2-7. Скрипт `MouseClick.au3` с выбором режима координат

```
1 Opt("MouseCoordMode", 2)
2 $hWnd = WinGetHandle("[CLASS:MSPaintApp]")
3 WinActivate($hWnd)
4 MouseClick("left", 250, 300)
```

Запустив этот скрипт, вы заметите, что координаты чёрной точки, нарисованной в окне Pain, изменились. Выбранный нами режим обеспечивает более точное позиционирование курсора. При разработке кликеров предпочтительнее использовать именно его. Он одинаково хорошо работает для окон в обычном и полноэкранном режимах. Единственный его недостаток заключается в сложности отладки скриптов. Утилиты вроде ColorPix отображают только абсолютные координаты пикселей.

Одно из распространённых действий мышью в компьютерных играх – перетаскивание (drag-and-drop). Для его симуляции AutoIt предоставляет функцию `MouseClickDrag`. Листинг 2-8 демонстрирует её использование.

Листинг 2-8. Скрипт `MouseClickDrag.au3`

```
1 $hWnd = WinGetHandle("[CLASS:MSPaintApp]")
2 WinActivate($hWnd)
3 MouseClickDrag("left", 250, 300, 400, 500)
```

После запуска скрипт `MouseClickDrag.au3` рисует линию в окне Paint. Координаты её начала: x = 250, y = 300. Она заканчивается в точке x = 400, y = 500. Функция AutoIt `MouseClickDrag` делает внутри себя уже знакомый нам WinAPI вызов `mouse_event`. Обе AutoIt функции `MouseClick` и `MouseClickDrag` симулируют действия мыши только в активном окне.

Действия мыши в неактивном окне

AutoIt предоставляет функцию `ControlClick`, которая симулирует щелчок мыши в неактивном окне. Пример её использования приведён в листинге 2-9.

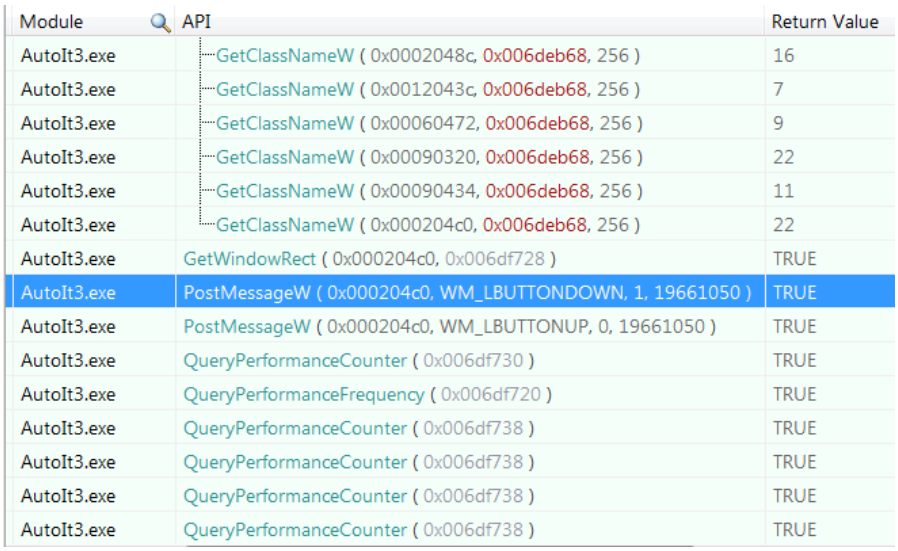
Листинг 2-9. Скрипт `ControlClick.au3`

```
1 $hWnd = WinGetHandle("[CLASS:MSPaintApp]")
2 ControlClick($hWnd, "", "Afx:00000000FFC20000:81", "left", 1, 250, 300)
```

Скрипт `ControlClick.au3` симулирует щелчок мыши в неактивном или свёрнутом окне Paint. По принципу работы функция `ControlClick` похожа на `ControlSend`. Вы должны указать элемент интерфейса по которому будет выполнен щелчок. В нашем случае – это рабочая область окна Paint, в которой пользователь применяет инструменты рисования (например кисти). Согласно информации от утилиты Au3Info, элемент рабочей области имеет класс “Afx:00000000FFC20000:81”.

Ради эксперимента мы можем передать одни и те же координаты курсора в функции `MouseClick` и `ControlClick`. В результате щелчки мыши произойдут в разных точках экрана. Причина в том, что входные параметры функции `ControlClick` – это координаты относительно левого верхнего угла указанного элемента интерфейса. В случае скрипта `ControlClick.au3`, щелчок произойдёт в точке x = 250, y = 300 относительно левого верхнего угла рабочей области. Тогда как режим координат для функции `MouseClick` определяется параметром `MouseCoordMode`.

`ControlClick` дважды вызывает WinAPI-функцию `PostMessageW` внутри себя. Иллюстрация 2-6 демонстрирует её вызовы, перехваченные с помощью API Monitor.



Module	API	Return Value
AutoIt3.exe	GetClassNameW (0x0002048c, 0x006deb68, 256)	16
AutoIt3.exe	GetClassNameW (0x0012043c, 0x006deb68, 256)	7
AutoIt3.exe	GetClassNameW (0x00060472, 0x006deb68, 256)	9
AutoIt3.exe	GetClassNameW (0x00090320, 0x006deb68, 256)	22
AutoIt3.exe	GetClassNameW (0x00090434, 0x006deb68, 256)	11
AutoIt3.exe	GetClassNameW (0x000204c0, 0x006deb68, 256)	22
AutoIt3.exe	GetWindowRect (0x000204c0, 0x006df728)	TRUE
AutoIt3.exe	PostMessageW (0x000204c0, WM_LBUTTONDOWN, 1, 19661050)	TRUE
AutoIt3.exe	PostMessageW (0x000204c0, WM_LBUTTONUP, 0, 19661050)	TRUE
AutoIt3.exe	QueryPerformanceCounter (0x006df730)	TRUE
AutoIt3.exe	QueryPerformanceFrequency (0x006df720)	TRUE
AutoIt3.exe	QueryPerformanceCounter (0x006df738)	TRUE
AutoIt3.exe	QueryPerformanceCounter (0x006df738)	TRUE
AutoIt3.exe	QueryPerformanceCounter (0x006df738)	TRUE
AutoIt3.exe	QueryPerformanceCounter (0x006df738)	TRUE

Иллюстрация 2-6. Внутренние вызовы функции `ControlClick`

При вызове функции `PostMessageW` первый раз, в неё передаётся параметр `WM_LBUTTONDOWN`. В результате симулируется нажатие кнопки мыши и её удержание. Во втором вызове передаётся параметр `WM_LBUTTONUP`, что соответствует отпусканию кнопки мыши.

Функция `ControlClick` работает ненадёжно со свёрнутыми окнами DirectX. В некоторых случаях щелчок мыши полностью игнорируется. Иногда он отрабатывает не в момент вызова `ControlClick`, а только после того, как свёрнутое окно будет восстановлено.

Выводы

Мы рассмотрели функции AutoIt, которые позволяют симулировать наиболее распространённые действия клавиатуры и мыши в окне игрового приложения. Эти функции делятся на два типа. Первый тип симулирует действия устройства только в активном окне. Второй тип работает как с активными, так и с неактивными или свёрнутыми окнами. Главный недостаток функций второго типа – недостаточная надёжность, поскольку некоторые приложения игнорируют симулируемые ими действия. Поэтому для реализации кликеров рекомендуется использовать функции первого типа.

Перехват устройств вывода

В этом разделе мы познакомимся с методами перехвата данных с устройств вывода. Сначала мы изучим, какие возможности Windows предоставляет приложениям для работы с этими устройствами. Затем рассмотрим способы перехвата выводимых на них изображений.

Интерфейс графических устройств Windows

Интерфейс графических устройств (Graphics Device Interface или GDI) – один из основных компонентов Windows, который отвечает за представление графических объектов и передачу их на устройства вывода. Обычно все элементы интерфейса окна приложения создаются с помощью графических объектов, таких как **контекст устройства** (device context или DC), **битовое изображение** (bitmap), кисти, цвета, шрифты.

Ключевая концепция GDI – это контекст устройства. Он представляет собой абстракцию, благодаря которой разработчики могут единообразно работать с графическими объектами независимо от устройства вывода (монитором, принтером, плоттером или графопостроителем и т.д). Сначала все операции по подготовке изображения выполняются над контекстом устройства в памяти. Затем готовый результат отправляется на устройство вывода.

На иллюстрации 2-7 приведены два контекста устройств, которые содержат изображения окон двух приложений A и B. Также на ней представлен DC, соответствующий итоговому изображению всего рабочего стола. ОС может собрать это изображение из всех видимых окон

и визуальных элементов рабочего стола (например панели задач). Когда контекст устройства подготовлен в памяти, ОС выводит его содержимое на экран.

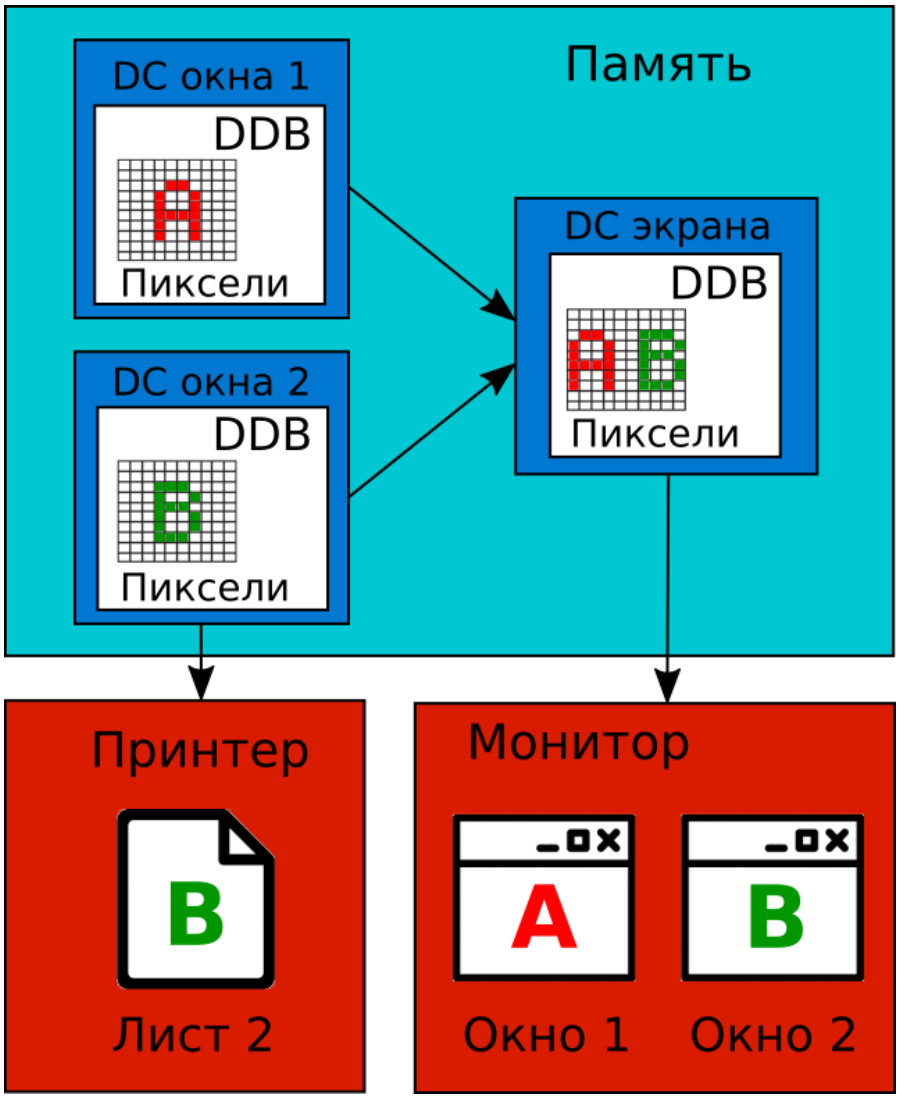


Иллюстрация 2-7. Отображение графических объектов на устройства вывода

Предположим, вам нужно напечатать документ, открытый в текстовом редакторе (окно B). В этом случае ОС просто отправляет DC окна этого приложения на принтер. Контексты устройств, связанные с другими открытыми в данный момент окнами игнорируются.

Контекст устройства представляет собой структуру в памяти. Разработчики могу работать с ней только через WinAPI-функции. Каждый DC содержит **аппаратно-зависимое битовое изображение** (Device Depended Bitmap или DDB). Битовое изображение – это представление поверхности для рисования в памяти. Все операции над графическими объектами в контексте устройства отражаются на соответствующем битовом изображении. Следовательно, оно хранит результат всех этих операций.

Битовое изображение состоит из двух основных частей:

1. Массив битов, описывающих наименьшие логические элементы изображения, которые называются **пикселями**.
2. Метаинформация.

У каждого пикселя есть два параметра: координаты и цвет. Их соответствие задаётся двумерным массивом. Номера элементов массива (индексы) равны координатам пикселя по осям X и Y. Числовое значение элемента массива соответствует коду цвета в палитре, которая связана с данным битовым изображением. Для анализа изображения все элементы двумерного массива должны обрабатываться последовательно.

Когда изображение подготовлено в контексте устройства, оно передаётся на настоящее устройство вывода. Как правило, функции системных библиотек выполняют необходимые преобразования изображения. Например, библиотека `vga.dll` подготавливает его для вывода на экран. Благодаря им драйвер устройства получает картинку в удобном для него формате.

Функции AutoIt для анализа изображений

AutoIt предоставляет функции для анализа текущего изображения на экране. Все они оперируют объектами GDI. Сейчас мы подробно рассмотрим эти функции.

Анализ отдельного пикселя

Самая простая операция при анализе изображения – это чтение цвета одного пикселя. Для этого необходимо знать его координаты. AutoIt поддерживает несколько режимов координат, которые представлены в

таблице 2-3. Они идентичны режимам координат позиционирования курсора мыши из таблицы 2-2.

Таблица 2-3. Режимы координат функций анализа изображений AutoIt

Режим	Описание
0	Координаты относительно левой верхней точки активного окна.
1	Абсолютные координаты экрана. Это режим по умолчанию.
2	Координаты относительно левой верхней точки клиентской части окна (без заголовка, меню и границ).

Выбрать нужный режим координат можно с помощью функции `Opt`, вызванной с первым параметром `PixelCoordMode`. Например, следующий вызов переключает скрипт во второй режим:

```
1 Opt("PixelCoordMode", 2)
```

Функция AutoIt `PixelGetColor` читает цвет пикселя. Входными параметрами она принимает координаты X и Y пикселя. Функция возвращает код цвета в **десятичной системе счисления**. Листинг 2-10 демонстрирует её использование.

Листинг 2-10. Скрипт `PixelGetColor.au3`

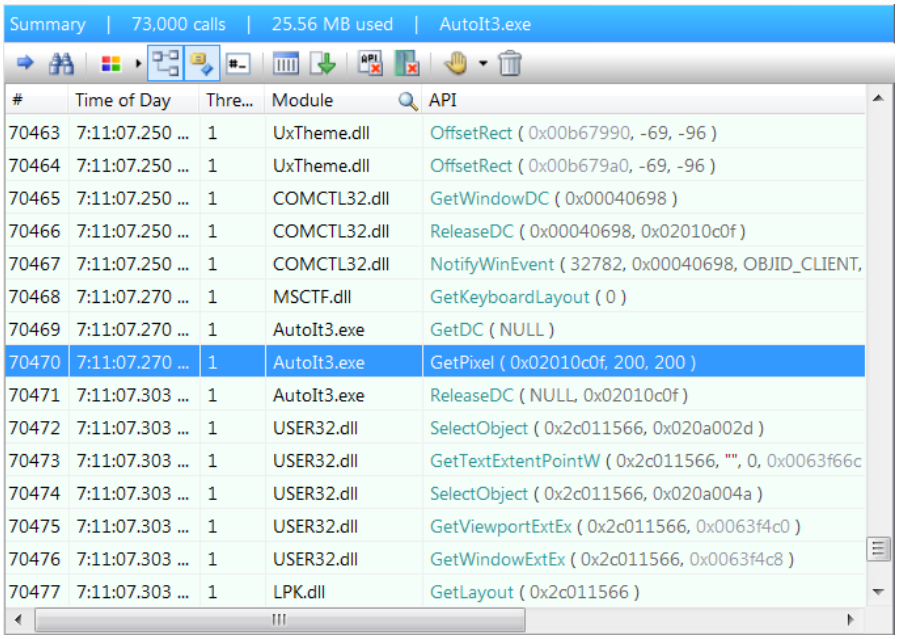
```
1 $color = PixelGetColor(200, 200)
2 MsgBox(0, "", "Цвет пикселя: " & Hex($color, 6))
```

Скрипт `PixelGetColor.au3` читает цвет пикселя с координатами x = 200, y = 200. После этого функция `MsgBox` выводит диалоговое окно с результатом. После запуска скрипта, вы увидите сообщение вроде: “Цвет пикселя 0355BB”.

Цвет кодируется числом 0355BB в **шестнадцатеричной системе счисления**. Такое представление широко распространено и называется **цветовой моделью RGB**. В ней любой цвет представляется координатами в трёхмерном **цветовом пространстве**. Трём его осям соответствуют цвета: X – красный (Red), Y – зелёный (Green) и Z – синий (Blue). Таким образом, цвет 0355BB из нашего примера соответствует точке с координатами: X = 03, Y = 55, Z = BB. Большинство графических редакторов и утилит используют этот способ кодирования.

Если вы переместите окно Notepad так, чтобы перекрыть им точку с координатой x = 200, y = 200 рабочего стола, результат возвращаемый скриптом `PixelGetColor.au3` изменится. Это означает, что он анализирует не конкретное окно, а изображение всего рабочего стола.

Иллюстрация 2-8 демонстрирует перехваченные WinAPI вызовы скрипта `PixelGetColor.au3`.



#	Time of Day	Thre...	Module	API
70463	7:11:07.250 ...	1	UxTheme.dll	OffsetRect (0x00b67990, -69, -96)
70464	7:11:07.250 ...	1	UxTheme.dll	OffsetRect (0x00b679a0, -69, -96)
70465	7:11:07.250 ...	1	COMCTL32.dll	GetWindowDC (0x00040698)
70466	7:11:07.250 ...	1	COMCTL32.dll	ReleaseDC (0x00040698, 0x02010c0f)
70467	7:11:07.250 ...	1	COMCTL32.dll	NotifyWinEvent (32782, 0x00040698, OBJID_CLIENT,
70468	7:11:07.270 ...	1	MSCTF.dll	GetKeyboardLayout (0)
70469	7:11:07.270 ...	1	AutoIt3.exe	GetDC (NULL)
70470	7:11:07.270 ...	1	AutoIt3.exe	GetPixel (0x02010c0f, 200, 200)
70471	7:11:07.303 ...	1	AutoIt3.exe	ReleaseDC (NULL, 0x02010c0f)
70472	7:11:07.303 ...	1	USER32.dll	SelectObject (0x2c011566, 0x020a002d)
70473	7:11:07.303 ...	1	USER32.dll	GetTextExtentPointW (0x2c011566, "", 0, 0x0063f66c
70474	7:11:07.303 ...	1	USER32.dll	SelectObject (0x2c011566, 0x020a004a)
70475	7:11:07.303 ...	1	USER32.dll	GetViewportExtEx (0x2c011566, 0x0063f4c0)
70476	7:11:07.303 ...	1	USER32.dll	GetWindowExtEx (0x2c011566, 0x0063f4c8)
70477	7:11:07.303 ...	1	LPK.dll	GetLayout (0x2c011566)

Иллюстрация 2-8. WinAPI вызовы скрипта `PixelGetColor.au3`

Функция `PixelGetColor` делает внутри себя три WinAPI вызова в следующей последовательности: `GetDC`, `GetPixel`, `ReleaseDC`. `GetDC` получает входным параметром значение “NULL”. Таким образом мы выбираем контекст устройства всего экрана для дальнейших операций. Если мы передадим в функцию `GetDC` дескриптор окна, мы получим DC его клиентской области. Благодаря этому наш скрипт сможет анализировать неактивные или перекрытые окна.

Дескриптор окна можно передать третьим параметром в AutoIt функцию `PixelGetColor`. Листинг 2-11 демонстрирует это решение.

Листинг 2-11. Скрипт `PixelGetColorWindow.au3`

```
1 $hWnd = WinGetHandle("[CLASS:MSPaintApp]")
2 $color = PixelGetColor(200, 200, $hWnd)
3 MsgBox(0, "", "Цвет пикселя: " & Hex($color, 6))
```

Скрипт `PixelGetColorWindow.au3` должен вернуть цвет пикселя в окне Paint, даже если оно неактивно. Мы ожидаем прочитать белый цвет с кодом “FFFFFF”, потому что область для рисования по умолчанию пуста.

Скрипт работает корректно, если окно Paint активно. Теперь попробуем перекрыть его окном другого приложения (например интерпретатором командной строки CMD). Скрипт прочитает чёрный цвет вместо белого.

Сравним WinAPI вызовы скриптов `PixelGetColorWindow.au3` и `PixelGetColor.au3`, перехваченные с помощью приложения API Monitor. В обоих случаях функция `GetDC` получает “NULL” входным параметром. Такое поведение похоже на ошибку в реализации функции `PixelGetColor` версии 3.3.14.1 AutoIt. Возможно, она будет исправлена в следующий версиях. Попробуем эту ошибку обойти.

Проблема функции `PixelGetColor` в некорректном параметре при вызове `GetDC`. Мы знаем, к каким WinAPI-функциям обращается `PixelGetColor`. Поэтому можем вызвать их напрямую из нашего скрипта, но с корректными параметрами. Результат приведён в листинге 2-12.

Листинг 2-12. Скрипт `GetPixel.au3`

```
#include <WinAPIGdi.au3>

$hWnd = WinGetHandle("[CLASS:MSPaintApp]")
$hDC = _WinAPI_GetDC($hWnd)
$color = _WinAPI_GetPixel($hDC, 200, 200)
MsgBox(0, "", "Цвет пикселя: " & Hex($color, 6))
```

Скрипт `GetPixel.au3` начинается с ключевого слова `include`. С его помощью мы включаем файл `WinAPIGdi.au3`, который содержит обёртки `_WinAPI_GetDC` и `_WinAPI_GetPixel` для соответствующих WinAPI-функций. Этот скрипт читает цвет пикселя окна Paint, независимо от того перекрыто оно или нет.

У рассмотренного нами решения есть одна проблема. Если вы свернёте окно Paint и запустите скрипт, он вернёт белый цвет. Этот результат выглядит корректным. Теперь попробуем изменить цвет рабочей области Paint, залив её для примера красным. Свернём окно снова, и запустим скрипт. Он опять прочитает белый цвет, хотя мы ожидаем красный. Рассмотрим, почему это происходит.

У каждого окна есть клиентская область. В этой области находятся все элементы интерфейса кроме заголовка окна, его границ и главного меню. Наша проблема с чтением цвета пикселя возникла из-за того, что размер клиентской области свёрнутого окна равен нулю. Следовательно, контекст устройства, связанный с окном, имеет пустое битовое изображение. При попытке чтения несуществующего пикселя, функция WinAPI `GetPixel` возвращает белый цвет.

Мы можем прочитать размер клиентской области окна с помощью скрипта, представленного в листинге 2-13.

Листинг 2-13. Скрипт `GetClientRect.au3`

```
#include <WinAPI.au3>

$hWnd = WinGetHandle("[CLASS:MSPaintApp]")
$tRECT = _WinAPI_GetClientRect($hWnd)
MsgBox(0, "Прямоугольник", _
       "Левый край: " & DllStructGetData($tRECT, "Left") & @CRLF & _
       "Правый край: " & DllStructGetData($tRECT, "Right") & @CRLF & _
       "Верхний край: " & DllStructGetData($tRECT, "Top") & @CRLF & _
       "Нижний край: " & DllStructGetData($tRECT, "Bottom"))
```

Скрипт `GetClientRect.au3` выводит X- и Y-координаты верхней левой и правой нижней точки клиентской области окна Paint. Если оно свёрнуто, все координаты равны нулю. В противном случае мы получим ненулевые числа.

Ограничение при работе со свёрнутым окном крайне неудобно, если вы планируете запускать бота и переключаться на другие приложения. У этой проблемы есть решение. Windows позволяет восстановить свёрнутое окно в прозрачном режиме. После этого можно скопировать битовое изображение его клиентской области в DC, связанный с оперативной памятью, и свернуть окно снова. Для копирования можно воспользоваться WinAPI-функцией `PrintWindow`. После можно анализировать копию с помощью известной нам AutoIt-обёртки `_WinAPI_GetPixel`.

Следующая статья подробно рассматривает работу со свёрнутыми окнами.

Анализ изменений картинки

Мы рассмотрели методы чтения цвета отдельно взятого пикселя. Однако, в большинстве случаев точные координаты нужного пикселя в окне игрового приложения неизвестны. Причина этого в том, что мы имеем не статическую картинку, а изображения движущихся игровых объектов.

Следовательно, мы должны найти способ анализа изменений на экране. AutoIt предоставляет несколько функций, подходящих для решения этой задачи.

Предположим, что мы ищем конкретный игровой объект на экране. Мы знаем его цвет, но не координаты. Эта задача является обратной той, которую решает функция AutoIt `PixelGetColor`. Для поиска координат игрового объекта по его цвету можно воспользоваться функцией `PixelSearch`. Листинг 2-14 демонстрирует пример.

Листинг 2-14. Скрипт `PixelSearch.au3`

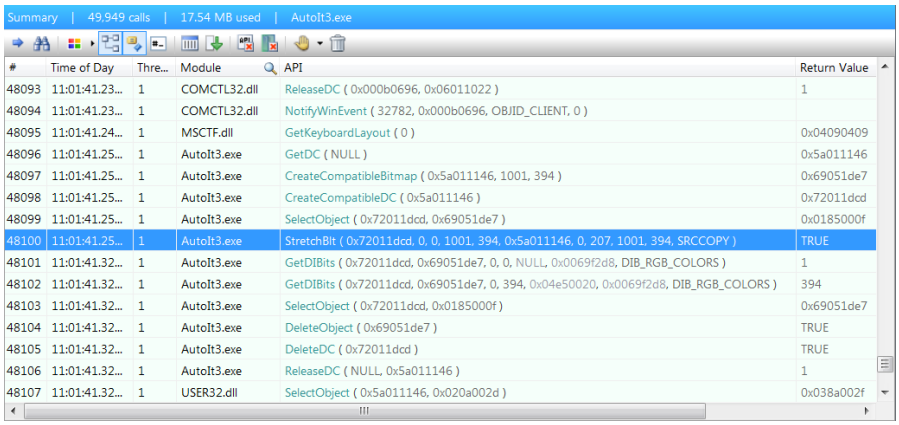
```
$coord = PixelSearch(0, 207, 1000, 600, 0x000000)
If @error = 0 then
    MsgBox(0, "", "Координата чёрной точки: x = " & $coord[0] & _
           " y = " & $coord[1])
else
    MsgBox(0, "", "Чёрная точка не найдена")
endif
```

Скрипт `PixelSearch.au3` ищет пиксель чёрного цвета с кодом 000000 в прямоугольной области экрана с координатами верхнего левого угла x = 0, y = 207 и правого нижнего – x = 1000, y = 600. Если в процессе поиска происходит ошибка, мы обрабатываем её с помощью **макроса** `@error`. В этом случае выводится сообщение: “Чёрная точка не найдена”.

Макрос `@error` можно рассматривать как глобальную переменную. Если в процессе работы AutoIt функции происходит ошибка, её код будет записан в `@error`. При обработке ошибки важно проверять макрос сразу после вызова функции, поскольку последующие вызовы могут переписать его значение.

Воспользуемся приложением Paint, чтобы протестировать скрипт `PixelSearch.au3`. Сначала поставим чёрную точку с помощью карандаша или кисти в области для рисования. Затем запустим скрипт. Он выведет координаты точки в диалоговом окне. Если этого не произошло, убедитесь, что Paint не перекрывают другие окна.

Проверим, какие вызовы WinAPI делает функция `PixelSearch`. Для этого запустим скрипт `PixelSearch.au3` из приложения API Monitor. Подождём, пока он отработает. После этого будем искать текст “0, 207” (координаты точки) в окне “Summary”. Вы должны найти вызов WinAPI `StretchBlt`, как показано на иллюстрации 2-9.



#	Time of Day	Thre...	Module	API	Return Value
48093	11:01:41.23...	1	COMCTL32.dll	ReleaseDC (0x000b0696, 0x06011022)	1
48094	11:01:41.23...	1	COMCTL32.dll	NotifyWinEvent (32782, 0x000b0696, OBJID_CLIENT, 0)	
48095	11:01:41.24...	1	MSCTF.dll	GetKeyboardLayout (0)	0x04090409
48096	11:01:41.25...	1	AutoIt3.exe	GetDC (NULL)	0x5a011146
48097	11:01:41.25...	1	AutoIt3.exe	CreateCompatibleBitmap (0x5a011146, 1001, 394)	0x69051de7
48098	11:01:41.25...	1	AutoIt3.exe	CreateCompatibleDC (0x5a011146)	0x72011dcd
48099	11:01:41.25...	1	AutoIt3.exe	SelectObject (0x72011dcd, 0x69051de7)	0x0185000f
48100	11:01:41.25...	1	AutoIt3.exe	StretchBlt (0x72011dcd, 0, 0, 1001, 394, 0x5a011146, 0, 207, 1001, 394, SRCCOPY)	TRUE
48101	11:01:41.32...	1	AutoIt3.exe	GetDIBits (0x72011dcd, 0x69051de7, 0, 0, NULL, 0x0069f2d8, DIB_RGB_COLORS)	1
48102	11:01:41.32...	1	AutoIt3.exe	GetDIBits (0x72011dcd, 0x69051de7, 0, 394, 0x04e50020, 0x0069f2d8, DIB_RGB_COLORS)	394
48103	11:01:41.32...	1	AutoIt3.exe	SelectObject (0x72011dcd, 0x0185000f)	0x69051de7
48104	11:01:41.32...	1	AutoIt3.exe	DeleteObject (0x69051de7)	TRUE
48105	11:01:41.32...	1	AutoIt3.exe	DeleteDC (0x72011dcd)	TRUE
48106	11:01:41.32...	1	AutoIt3.exe	ReleaseDC (NULL, 0x5a011146)	1
48107	11:01:41.32...	1	USER32.dll	SelectObject (0x5a011146, 0x020a002d)	0x038a002f

Иллюстрация 2-9. WinAPI-вызовы функции `PixelSearch`

Функция `StretchBlt` копирует битовое изображение из DC экрана в контекст устройства памяти, который также известен как **совместимый контекст устройства** (compatible device context). Чтобы проверить это предположение, сравним входные параметры вызовов `GetDC`, `CreateCompatibleBitmap`, `CreateCompatibleDC`, `SelectObject` и `StretchBlt` в окне API Monitor.

Функция `GetDC` возвращает дескриптор DC экрана, который в нашем случае равен 0x5a011146. Что означает это шестнадцатеричное число? Воспользуемся документацией WinAPI, чтобы уточнить определение типа `HDC`, соответствующее дескриптору DC:

```
1 typedef void *PVOID;
2 typedef PVOID HANDLE;
3 typedef HANDLE HDC;
```

`HDC` представляет собой указатель на область памяти. Следовательно, 0x5a011146 – это адрес памяти, где хранится дескриптор.

Вызов `CreateCompatibleBitmap` идёт после `GetDC`. Он создаёт битовое изображение для работы над ним в памяти. Первым входным параметром `CreateCompatibleBitmap` принимает дескриптор DC экрана. Далее с помощью `CreateCompatibleDC` создаётся совместимый контекст устройства. Вызовом `SelectObject` в него загружается битовое изображение. После этого вызов `StretchBlt` может выполнить копирование изображения из контекста экрана (дескриптор 0x5a011146) в совместимый DC в памяти.

На следующем шаге AutoIt-функции `PixelSearch` происходит WinAPI-вызов `GetDIBits`. Он конвертирует аппаратно-зависимое битовое изображение (DDB) в **аппаратно-независимое** (DIB). Зачем это нужно? DIB формат более удобен, поскольку позволяет работать с изображениями как с обычным массивом.

Заключительный шаг функции `PixelSearch` – проход по всем пикселям DIB и сравнение цвета каждого из них с заданным. Для этой операции вызовы WinAPI не нужны.

Пример C++ реализации захвата изображений с экрана доступен в WinAPI документации. Эта реализация демонстрирует копирование битового изображения в совместимый DC и преобразование DDB в DIB.

У функции `PixelSearch` есть необязательный пятый параметр, через который можно передать дескриптор окна. В этом случае поиск пикселя происходит именно в нём. Если параметр не указан, функция ищет на всём экране.

Листинг 2-15 демонстрирует поиск пикселя в заданном окне.

Листинг 2-15. Скрипт `PixelSearchWindow.au3`

```
$hWnd = WinGetHandle("[CLASS:MSPaintApp]")
$coord = PixelSearch(0, 207, 1000, 600, 0x000000, 0, 1, $hWnd)
If @error = 0 then
    MsgBox(0, "", "Координата чёрной точки: x = " & $coord[0] & _
           " y = " & $coord[1])
else
    MsgBox(0, "", "Чёрная точка не найдена")
endif
```

Согласно документации AutoIt, скрипт `PixelSearchWindow.au3` должен искать пиксель в перекрытом окне Paint, но этого не происходит. Похоже, что мы снова столкнулись с ошибкой, которая проявлялась ранее в функции `PixelGetColor`. API Monitor подтвердит, что в WinAPI вызов `GetDC` снова передаётся “NULL” вместо дескриптора окна. По этой причине `PixelSearch` всегда обрабатывает DC экрана, независимо от своего пятого параметра. Вы можете обойти эту ошибку, если будете работать с WinAPI напрямую. Пример аналогичного решения приведён в листинге 2-12. В этом случае вам необходимо полностью повторить алгоритм функции `PixelSearch`.

`PixelChecksum` – это ещё одна функция AutoIt для анализа движущихся изображений. Рассмотренные нами ранее функции `PixelGetColor` и `PixelSearch` позволяют получить информацию о единственном пикселе. `PixelChecksum` работает иначе. Она обнаруживает изменение изображения в заданной области экрана. Это может быть полезно, когда бот должен реагировать на игровые события.

Функция `PixelChecksum` рассчитывает **контрольную сумму** (checksum) для пикселей в указанной области. Эта сумма представляет собой число, полученное в результате применения определённого алгоритма к набору данных. Простейшим примером такого алгоритма может быть суммирование кодов цветов пикселей. Если цвет хотя бы одного пикселя поменяется, результирующая контрольная сумма также изменится.

Листинг 2-16 демонстрирует применение функции `PixelChecksum`.

Листинг 2-16. Скрипт `PixelChecksum.au3`

```
$checkSum = PixelChecksum(0, 0, 50, 50)

while $checkSum = PixelChecksum(0, 0, 50, 50)
    Sleep(100)
wend

MsgBox(0, "", "Изображение в области экрана изменилось")
```

Скрипт `PixelChecksum.au3` выводит диалоговое окно, если меняется изображение в области экрана между точками с координатами x = 0, y = 0 и x = 50, y = 50. В скрипте многократно вызывается функция `PixelChecksum`. Первый раз она вычисляет начальное значение контрольной суммы. После этого функция вызывается каждые 100 миллисекунд в цикле `while`. Временная задержка выполняется с помощью вызова `Sleep`. Цикл продолжается до тех пор, пока контрольная сумма не изменится. Как только это происходит, цикл прерывается и выводится диалоговое окно.

Рассмотрим внутренние вызовы функции `PixelChecksum`. API Monitor покажет нам ту же самую последовательность WinAPI вызовов, что и для функции `PixelSearch`. Это означает, что AutoIt следует одному и тому же алгоритму для получения DIB из изображения на экране. Однако, последний шаг этих двух функций отличается. `PixelChecksum` вычисляет контрольную сумму по указанному алгоритму. Вы можете выбрать один из

двух доступных алгоритмов: **ADLER** или **CRC32**. Рассмотрим их различия.

Любой алгоритм расчёта контрольных сумм имеет **коллизии**. Коллизия – это два разных набора входных данных, для которых функция возвращает одинаковый результат. Алгоритмы предлагаемые AutoIt отличаются скоростью и надёжностью. CRC32 работает медленнее чем ADLER, но имеет меньше коллизий. Следовательно, надёжность CRC32 выше и использующий его бот будет реже ошибаться.

Все рассмотренные AutoIt функции для анализа пикселей работают в полноэкранных окнах DirectX приложений. Вы можете использовать их для разработки своих ботов без каких либо ограничений.

Библиотеки для анализа изображений

Мы рассмотрели средства AutoIt для анализа изображений на экране. Кроме них есть более мощные функции, предоставляемые сторонними библиотеками. Рассмотрим их подробнее.

Библиотека FastFind

Библиотека FastFind предоставляет мощные функции для анализа изображений, которые хорошо подходят для поиска игровых объектов на экране. Эти функции доступны как из AutoIt скриптов, так и из C++ приложений.

Для вызова функции библиотеки из AutoIt скрипта выполните следующие шаги:

1. Создайте отдельную папку для вашего скрипта. Для примера назовём её `FFDemo`.
2. Скопируйте файл `FastFind.au3` из архива FastFind библиотеки в папку `FFDemo`.
3. Скопируйте также один из файлов `FastFind.dll` или `FastFind64.dll`. Если вы работаете на 64-битной версии Windows, вам нужен файл `FastFind64.dll`, иначе – `FastFind.dll`.
4. Включите файл `FastFind.au3` в ваш скрипт с помощью `include`:

```
1 #include "FastFind.au3"
```

Теперь вы можете вызывать функции FastFind в своём скрипте.

Для работы с функциями библиотеки из C++ приложения сделайте следующее:

1. Скачайте и установите компилятор C++. Это может быть IDE **Visual Studio Community** с сайта Microsoft, в которую уже встроен компилятор. Альтернативным решением является набор инструментов **MinGW**.
2. Если вы используете MinGW, создайте файл с исходным кодом (например `test.cpp`). В случае Visual Studio, создайте проект “Win32 Console Application”.
3. Скопируйте код из листинга 2-17 в свой CPP-файл.
4. Скопируйте из архива библиотеки файл `FastFind.dll` в папку вашего проекта. `FastFind64.dll` следует копировать только в том случае, если вы собираетесь компилировать 64-битные исполняемые файлы.
5. Если вы используете MinGW, создайте файл с именем `Makefile` и следующим содержанием:

```
all:
    g++ test.cpp -o test.exe
```

6. В случае использования MinGW, скомпилируйте приложение с помощью команды `make`, запущенной в командной строке CMD. Для Visual Studio достаточно нажать горячую клавишу F7.

Листинг 2-17. Файл `test.cpp`

```
#include <iostream>

#define WIN32_LEAN_AND_MEAN
#include <windows.h>

using namespace std;

typedef LPCTSTR(CALLBACK* LPFNDLLFUNC1)(void);

HINSTANCE hDLL;               // Дескриптор DLL-библиотеки
LPFNDLLFUNC1 lpfnDllFunc1;    // Указатель на функцию
LPCTSTR uReturnVal;

int main()
{
    hDLL = LoadLibraryA("FastFind");
    if (hDLL != NULL)
    {
        lpfnDllFunc1 = (LPFNDLLFUNC1)GetProcAddress(hDLL, "FFVersion");
        if (!lpfnDllFunc1)
        {
            // Обработка ошибки
```

```
            FreeLibrary(hDLL);
            cout << "error" << endl;
            return 1;
        }
        else
        {
            // Вызов функции из библиотеки
            uReturnVal = lpfnDllFunc1();
            cout << "version = " << uReturnVal << endl;
        }
    }
    return 0;
}
```

После компиляции вы получите исполняемый EXE-файл. Запустив его, вы увидите вывод версии библиотеки FastFind в консоль:

```
version = 2.2
```

В нашем примере мы использовали **явную компоновку библиотеки** (explicit library linking) для доступа к её функциям. Также возможно альтернативное решение – **неявная компоновка библиотеки** (implicit library linking). Вы можете применять любой подход для работы с FastFind. Но во втором случае вам придётся использовать тот же компилятор C++ (желательно и той же версии), что и разработчики FastFind.

Какие задачи мы сможем решить с помощью библиотеки? Прежде всего, у нас появился более надёжный метод поиска игрового объекта. Функция `FFBestSpot` ищет область экрана, которая содержит максимальное число пикселей заданного цвета. Рассмотрим пример её использования.

На иллюстрации 2-10 приведён снимок экрана (или **скриншот**) популярной MMORPG-игры Lineage 2. На нём вы видите модели двух персонажей. В правой части расположен персонаж игрока с именем “Zagstruk”. Слева от него находится монстр “Wretched Archer”. Чтобы определить его координаты, применим функцию `FFBestSpot`.

Сначала нам нужно выбрать подходящий цвет для поиска. Лучше всего для этой цели подойдёт цвет текста над персонажами. При их перемещении, эти надписи не меняют свою геометрию. Также они не зависят от световых эффектов, приближения и угла поворота камеры. В этом случае поиск функцией `FFBestSpot` будет достаточно надёжным. Монстр, в отличие от игрока, имеет дополнительный текст зелёного цвета. Именно его мы и будем искать.

Иллюстрация 2-10. Скриншот известной MMORPG-игры Lineage 2

В некоторых случаях для поиска у нас нет статичных элементов интерфейса, таких как надписи над игровыми объектами. Тогда приходится искать модели персонажей. Функция `FFBestSpot` может оказаться недостаточно надёжной для этой задачи и часто давать ошибочный результат. Причина заключается в том, что тени и световые эффекты могут менять цвета моделей.

Листинг 2-18 демонстрирует поиск текста зелёного цвета с помощью функции `FFBestSpot`.

Листинг 2-18. Скрипт `FFBestSpot.au3`

```
#include "FastFind.au3"

Sleep(5 * 1000)

const $sizeSearch = 80
const $minNbPixel = 50
const $optNbPixel = 200
const $posX = 700
const $posY = 380

$coords = FFBestSpot($sizeSearch, $minNbPixel, $optNbPixel, _
                     $posX, $posY, 0xA9E89C, 10)

if not @error then
    MsgBox(0, "Coords", $coords[0] & ", " & $coords[1])
else
    MsgBox(0, "Coords", "Текст не найден")
endif
```

Если вы запустите скрипт `FFBestSpot.au3` и переключитесь на окно с иллюстрацией 2-10, появится диалоговое окно с координатами текста. После старта, скрипт ждёт пять секунд, в течение которых вы должны переключиться на скриншот игры. Функция `FFBestSpot` отработает после этой задержки. Таблица 2-4 описывает её входные параметры.

Таблица 2-4. Входные параметры функции `FFBestSpot`

Параметр	Описание
sizeSearch	Ширина и высота квадратной области экрана для поиска.
minNbPixel	Минимальное число пикселей, которое должно быть в искомой области.
optNbPixel	Оптимальное число пикселей, которое должно быть в искомой области.
posX	Примерная координата X искомой области.
posY	Примерная координата Y искомой области.
0xA9E89C	Искомый цвет в шестнадцатеричной системе счисления.
10	Допустимое отклонение цвета от каждого из основных цветов (красный, зелёный, синий). Значение должно быть в диапазоне от 0 до 255.

Функция `FFBestSpot` возвращает массив из трёх элементов, если ей удаётся найти указанную область. В противном случае возвращается ноль и выставляется макрос `@error` с кодом ошибки. Первые два элемента массива с результатом – это X и Y координаты найденной области. Третий элемент равен числу пикселей указанного цвета в ней. Более подробную информацию о функции вы можете найти в файле документации `FastFind.chm` из архива библиотеки.

Функция `FFBestSpot` хорошо подходит для поиска элементов интерфейса, таких как индикатор здоровья, иконки, окна и текст. Кроме того, с её помощью можно успешно искать игровые объекты в 2D играх.

Вторая задача, которую хорошо решает FastFind, заключается в обнаружении изменений изображения на экране. Функция `FFLocalizeChanges` реализует подходящий алгоритм. Для демонстрации её работы воспользуемся окном приложения Notepad.

Скрипт `FFLocalizeChanges.au3`, приведённый в листинге 2-19, определяет координаты текста, который вы введёте в окне Notepad.

Листинг 2-19. Скрипт `FFLocalizeChanges.au3`

```
#include "FastFind.au3"

Sleep(5 * 1000)
FFSnapShot (0, 0, 0, 0, 0)

MsgBox(0, "Info", "Измените изображение")

Sleep(5 * 1000)
FFSnapShot (0, 0, 0, 0, 1)

$coords = FFLocalizeChanges(0, 1, 10)

if not @error then
    MsgBox(0, "Coords", "x1 = " & $coords[0] & ", y1 = " & $coords[1] & _
           " x2 = " & $coords[2] & ", y2 = " & $coords[3])
else
    MsgBox(0, "Coords", "Изменения не обнаружены")
endif
```

Для тестирования скрипта `FFLocalizeChanges.au3` выполните следующие шаги:

1. Запустите приложение Notepad и разверните его окно на весь экран.
2. Запустите скрипт.
3. Переключитесь на окно Notepad.
4. Ожидайте диалоговое окно с сообщением “Измените изображение”.
5. Введите несколько символов в Notepad в течение пяти секунд.
6. Ожидайте диалоговое окно с координатами введённого текста.

Функции библиотеки FastFind оперируют абстракцией **SnapShot** (снимок). SnapShot – это копия текущего изображения на экране в память. По сути такой снимок очень похож на DIB. Когда мы использовали функцию `FFBestSpot`, она создавала SnapShot неявно. Затем на нём отрабатывал алгоритм поиска нужной области.

Функция `FFLocalizeChanges` принимает входными параметрами два SnapShot: до и после изменения. Она не знает, в какой момент времени произошло изменение. Поэтому SnapShot должен создавать пользователь библиотеки с помощью функции `FFSnapShot`. Получившиеся снимки будут сохранены в массиве, индексы которого начинаются с нуля. По умолчанию после каждого вызова `FFSnapShot` индекс инкриминируется. Но его можно указать и явно в пятом параметре функции. Первые четыре параметра `FFSnapShot` – это координаты X и Y верхнего левого и правого нижнего углов сохраняемой области. Если все координаты равны нулю, скопировано будет изображение всего экрана.

Рассмотрим алгоритм скрипта `FFLocalizeChanges.au3`. После пятисекундной задержки вызывается функция `FFSnapShot`, которая создаёт SnapShot экрана с первоначальным изображением окна Notepad. Затем выводится сообщение “Измените изображение”, после которого пользователь вводит текст. Спустя пять секунд, скрипт делает ещё один SnapShot. Оба SnapShot передаются в функцию `FFLocalizeChanges`, которая вычисляет координаты изменившейся области.

Входные параметры `FFLocalizeChanges` приведены в таблице 2-5.

Таблица 2-5. Входные параметры функции `FFLocalizeChanges`

Параметр	Описание
0	Индекс первого SnapShot для сравнения.
1	Индекс второго SnapShot для сравнения.
10	Допустимое отклонение цвета. Этот параметр работает так же, как и для функции `FFBestSpot`.

Функция `FFLocalizeChanges` возвращает массив из пяти элементов. Первые четыре из них – это координаты X и Y верхнего левого и правого нижнего углов изменённой области. Пятый элемент хранит число отличающихся пикселей. `FFLocalizeChanges` представляет собой хорошую альтернативу AutoIt функции `PixelChecksum`, потому что реже ошибается и предоставляет больше информации об обнаруженном изменении.

Функции библиотеки FastFind работают с перекрытыми окнами, но не со свёрнутыми. Большинству из них можно передать дескриптор окна через необязательный входной параметр. Также все функции работают корректно с полноэкранными окнами DirectX-приложений.

Библиотека ImageSearch

Библиотека ImageSearch решает одну единственную задачу. Она ищет заданный фрагмент изображения в указанной области экрана.

Для вызова функций библиотеки из AutoIt скрипта выполните следующие шаги:

1. Создайте папку для проекта (например с именем `ImageSearchDemo`).
2. Скопируйте в неё файлы `ImageSearch.au3` и `ImageSearchDLL.dll` из архива библиотеки.
3. Включите файл `ImageSearch.au3` в ваш скрипт:

```
1 #include "ImageSearch.au3"
```

После этого все функции библиотеки станут доступны.

Если вы разрабатываете приложение на C++ и планируете работать с ImageSearch, необходимо выполнить явную компоновку библиотеки. Пример этого метода приведён в предыдущем разделе, посвящённом FastFind.

Для демонстрации возможностей ImageSearch напишем скрипт для поиска иконки приложения Notepad на экране. Для начала подготовим фрагмент изображения, который будем искать. В нашем случае это иконка, приведённая на иллюстрации 2-11.

Иллюстрация 2-11. Иконка Notepad

Вы можете создать эту иконку с помощью приложения Paint. Для этого запустите приложение Notepad, сделайте скриншот окна, вставьте его в Paint и вырежьте иконку. Сохраните результат в файл с именем `notepad-logo.bmp` в папку с проектом `ImageSearchDemo`.

Листинг 2-20 демонстрирует скрипт `Search.au3` для поиска иконки на экране.

Листинг 2-20. Скрипт `Search.au3`

```
1 #include <ImageSearch.au3>
2
3 Sleep(5 * 1000)
4
5 global $x = 0, $y = 0
```

```
$search = _ImageSearch('notepad-logo.bmp', 0, $x, $y, 20)

if $search = 1 then
    MsgBox(0, "Coords", $x & ", " & $y)
else
    MsgBox(0, "Coords", "Фрагмент изображения не найден")
endif
```

Чтобы протестировать скрипт, выполните следующие шаги:

1. Запустите приложение Notepad.
2. Запустите скрипт `Search.au3`.
3. Сделайте активным окно Notepad.
4. Ожидайте сообщения с координатами иконки.

Если у вас возникли проблемы с последними версиями библиотеки, вы можете воспользоваться более старой, но стабильной сборкой.

Параметры функции `_ImageSearch` приведены в таблице 2-6.

Таблица 2-6. Входные параметры функции `_ImageSearch`

Параметр	Описание
notepad-logo.bmp	Путь к файлу с фрагментом изображения для поиска.
0	Флаг для выбора точки, координаты которой вернёт функция. Значение 0 соответствует верхнему левому углу фрагмента. Значение 1 – координатам его центра.
x	Переменная для записи X-координаты найденного фрагмента.
y	Переменная для Y-координаты.
20	Допустимое отклонение цвета.

В случае успешного поиска, функция возвращает нулевое значение. Если же произошла ошибка, возвращается её код.

Функция `_ImageSearch` ищет фрагмент изображения на всём экране. Библиотека также предоставляет функцию `_ImageSearchArea` для поиска только в указанной области экрана. Пример её вызова выглядит следующим образом:

```
$search = _ImageSearchArea('notepad-logo.bmp', 0, 100, 150, 400, 450, $x, $y, 20)
```

Четыре дополнительных параметра функции (со второго по шестой) – это координаты области экрана для поиска. В нашем примере она ограничена точками x = 100, y = 150 и x = 400, y = 450. `_ImageSearchArea` возвращает такой же результат, как и функция `_ImageSearch`: код ошибки и координаты найденного фрагмента через седьмой и восьмой входной параметр.

Функции библиотеки ImageSearch работают только с текущим изображением на экране. Это значит, что вы не можете перекрыть или свернуть окно анализируемого приложения. Полноэкранные окна DirectX-приложений обрабатываются корректно.

Библиотека ImageSearch – это надёжный инструмент для поиска статичных фрагментов в окне игрового приложения. Она хорошо подходит для обнаружения элементов интерфейса и 2D-объектов.

Выводы

Мы рассмотрели функции AutoIt для анализа пикселей изображения на экране и для обнаружения изменений этого изображения.

Мы изучили основные возможности библиотек FastFind и ImageSearch. Первая из них предоставляет более мощные функции анализа пикселей. Вторая позволяет найти фрагмент изображения на экране.

Пример кликера для Lineage 2

Напишем простого бота-кликера для MMORPG Lineage 2, чтобы закрепить полученные знания о техниках внедрения данных на уровне ОС и перехвате устройств вывода.

Обзор игры Lineage 2

Игровой процесс Lineage 2 типичен для жанра RPG. Сначала надо выбрать расу и класс для своего персонажа. Для получения новых умений и покупки предметов игрок должен выполнять задания (или **квесты**) и охотиться на монстров. Этот процесс получения ресурсов называется **фарминг** (farming). При этом у игроков всегда есть возможность общаться и взаимодействовать между собой, как и в любой MMORPG. Они могут помогать или мешать друг другу. Если несколько игроков хотят получить один и тот же ресурс, они должны сражаться за него. Этот элемент

соперничества представляет наиболее привлекательную часть игрового процесса. Поэтому пользователи стремятся как можно быстрее и лучше развить своего персонажа, чтобы сражаться между собой.

Самый прямолинейный путь развития персонажа – это охота на монстров. После убийства каждого из них, игрок получает очки опыта для улучшения умений персонажа, а также золото для покупки новых предметов. Мы попытаемся автоматизировать именно этот процесс, поскольку он ведёт к разностороннему развитию героя. Однако, есть и другие пути получения игровых ресурсов: торговля, рыбалка, создание предметов и выполнение заданий.

На иллюстрации 2-12 приведён скриншот игры. Рассмотрим на нём элементы игрового интерфейса, помеченные номерам:

1. Окно состояния с параметрами персонажа игрока. К наиболее важным из них относятся очки здоровья (health points или HP) и мана (mana points или MP).
2. Окно цели с информацией о выделенном в данный момент монстре. В нём есть полоска с HP цели.
3. Панель горячих клавиш с иконками возможных действий и доступных умений.
4. Окно чата для ввода команд и отправки сообщений другим игрокам.

Иллюстрация 2-12. Интерфейс Lineage 2

Тщательное изучение интерфейса поможет вам разработать наиболее простой и эффективный алгоритм взаимодействия бота с игрой. Более

подробно интерфейс Lineage 2 описан на вики-странице.

В интернете есть множество серверов Lineage 2. Они отличаются версией игры, дополнительными возможностями и системами защиты, которые предотвращают использование ботов. Наиболее эффективная защита работает на официальных серверах, которые поддерживают разработчики игры. Кроме них есть так называемые пиратские сервера, которые поддерживаются энтузиастами. Как правило, их защита значительно слабее. В нашем примере мы будем подключаться к серверу РПГ-Клуб.

Реализация бота

Чтобы лучше понять механику игры, попробуйте зарегистрироваться на сервере РПГ-Клуб, создать персонажа и убить нескольких монстров. Вы заметите, что почти всё время нажимаете одни и те же кнопки на панели горячих клавиш.

Теперь составим список действий, которые надо автоматизировать. Предлагаю следующий вариант:

1. Выбрать монстра для атаки. Это можно сделать двумя способами: левым щелчком мыши по нему или ввести в окно чата команду “/target”. Например:

```
/target ИмяМонстра
```

Полный список игровых команд приведён на официальном сайте. Их можно комбинировать в одно действие с помощью макросов.

2. Атаковать монстра. Для этого можно нажать кнопку “атака” на панели горячих клавиш или горячую клавишу F1.
3. Ожидать пока персонаж убьёт монстра.
4. Подобрать выпавшие из монстра предметы и золото. Опять же можно щёлкнуть мышью по действию на панели горячих клавиш или нажать F8.

Рассмотренные нами действия выглядят достаточно просто и прямолинейно. По сути у нас получился алгоритм работы бота. Напишем скрипт, который будет по нему работать.

Слепой бот

Начнём с того, что будем строго следовать нашему алгоритму охоты на монстров. На каждом его шаге бот должен симулировать нажатие одной клавиши. Такой кликер можно считать слепым, поскольку он не получает никакой информации о состоянии игровых объектов.

Перед тем как начать писать код, рассмотрим конфигурацию панели горячих клавиш. Вам нужно настроить её так же как на иллюстрации 2-13.

Иллюстрация 2-13. Панель горячих клавиш

Таблица 2-7 описывает конфигурацию панели.

Таблица 2-7. Действия и соответствующие им горячие клавиши

Клавиша	Действие
F1	Атаковать выделенного в данный момент монстра.
F2	Использовать наступательное умение по текущей цели.
F5	Использовать зелье лечения для восстановления HP.
F8	Подобрать с земли предметы, лежащие около персонажа.
F9	Макрос с командой `/target ИмяМонстра` для выбора цели.
F10	Выбор ближайшего монстра.

Теперь стало очевидно, как надо связать горячие клавиши с шагами алгоритма бота. Скрипт `BlindBot.au3`, приведённый в листинге 2-21, демонстрирует это.

Листинг 2-21. Скрипт `BlindBot.au3`

```
#RequireAdmin

Sleep(2000)

while true
    Send("{F9}")
    Sleep(200)
    Send("{F1}")
    Sleep(5000)
    Send("{F8}")
    Sleep(1000)
wend
```

В первой строчке скрипта стоит ключевое слово `#RequireAdmin`. Благодаря ему при старте скрипт потребует предоставить ему права администратора. Получив эти права, он сможет взаимодействовать с другими приложениями независимо от того, какой пользователь их запустил. Некоторые клиенты Lineage 2 при старте также требуют прав администратора. Поэтому к ним не смогут получить доступ скрипты AutoIt, запущенные от имени пользователя с меньшими правами. Я рекомендую всегда использовать `#RequireAdmin` в ваших кликерах.

Скрипт начинает своё выполнение с двухсекундной задержки. Она нужна для того, чтобы вы успели переключиться на окно Lineage 2. Текущая версия бота работает только с активным окном игры.

После вызова `Sleep` идёт бесконечный цикл `while`, в котором выполняются все действия бота:

1. `Send("{F9}")` – выбрать монстра с помощью макроса, настроенного на клавишу F9.
2. `Sleep(200)` – подождать 200 миллисекунд. Это время требуется клиенту Lineage 2, чтобы выделить монстра и отрисовать окно цели.

Помните, что все действия в окне игрового приложения происходят не мгновенно. Зачастую время на их выполнение намного меньше скорости реакции человека, поэтому вы его не замечаете. Но оно есть.

3. `Send("{F1}")` – атаковать выбранного монстра.
4. `Sleep(5000)` – ожидать пять секунд, пока персонаж не подбежит к монстру и не убьёт его.
5. `Send("{F8}")` – подобрать один выпавший предмет.

5. `Sleep(1000)` – ждать одну секунду, пока персонаж подбирает предмет.

В нашем примере последовательность действий бота строго определена. Поэтому каждое действие может завершиться успешно только в том случае, если предыдущее также было успешно. Это значит, что макрос выбора монстра должен отработать правильно. Если первый шаг не удался, все дальнейшие действия не имеют смысла. Затем персонаж

должен успеть подбежать к монстру и убить его за пять секунд. Очевидно, это время может меняться в зависимости от расстояния до цели. Наконец, бот ожидает, что из монстра выпадет только один предмет. Наш скрипт отработает правильно только тогда, когда все перечисленные условия выполнятся, иначе неизбежны ошибки.

Попробуйте запустить скрипт и проверить его работу. Часто бот будет совершать не те действия, которые нужны в данный момент. Причина в том, что одно из условий его работы нарушено. С другой стороны, все его ошибки не критичны, поскольку он продолжает свою работу. Это возможно благодаря особенности команды `/target` и механизму атаки цели. Если выполнить макрос `/target` дважды, бот будет атаковать уже выбранного монстра. Таким образом он всегда будет добивать цель. Даже если монстр выжил после первой итерации цикла `while`, атака на него продолжится в следующих итерациях. Кроме того, команда “поднять предмет” не прерывает атаку, если поблизости от персонажа нет предметов. Поэтому он будет продолжать бить цель и после пятисекундной задержки, отведённой на убийство монстра.

Единственная проблема, которую бот не сможет решить, заключается в подбирании выпадающих предметов. Число их случайно и зависит от вида монстра. Поэтому иногда они будут оставаться лежать на земле и персонаж недополучит свои ресурсы. В такой ситуации повторение действия “поднять” несколько раз будет лучшим, что можно придумать без дополнительных проверок. Даже если зачастую число нажатий будет больше необходимого, персонаж подберёт все выпавшие ресурсы.

Можно сделать скрипт более удобным для чтения и модификации, если вынести каждый шаг алгоритма в отдельную функцию с говорящим названием. Результат такого улучшения приведён в скрипте `BlindBotFunc.au3` из листинга 2-22.

Листинг 2-22. Скрипт `BlindBotFunc.au3`

```
#RequireAdmin

func SelectTarget()
    Send("{F9}")
    Sleep(200)
endfunc

func Attack()
    Send("{F1}")
    Sleep(5000)
endfunc

```

```
13 func Pickup()
14     Send("{F8}")
15     Sleep(1000)
16 endfunc
17
18 Sleep(2000)
19
20 while true
21     SelectTarget()
22     Attack()
23     Pickup()
24 wend
```

Теперь скрипт выглядит намного понятнее. Он начинает свою работу с вызова `Sleep(2000)`. Выше этой строчки находятся только объявления **пользовательских функций**, которые определены разработчиком для своих целей. Их код будет выполнен только в местах вызова, то есть в цикле `while`. Обратите внимание, что несмотря на изменившуюся структуру кода, алгоритмы скриптов `BlindBotFunc.au3` и `BlindBot.au3` остались идентичны.

Бот с условиями

Попробуем улучшить нашего бота и сделать его более эффективным. Он будет реже ошибаться, если сможет проверять результат каждого своего действия. Применим функцию анализа пикселей для чтения состояния окружающих его игровых объектов.

Перед тем как мы продолжим, было бы полезно добавить к текущей реализации бота механизм вывода диагностических сообщений. Техника вывода сообщений в местах принятия программой важных решений известна как **трассировка** (tracing). С её помощью мы сможем отследить, какие решения принимает бот в ходе своей работы.

Реализация функции вывода сообщений в файл представлена в листинге 2-23.

Листинг 2-23. Реализация функции `LogWrite`

```
1 global const $LogFile = "debug.log"
2
3 func LogWrite($data)
4     FileWrite($LogFile, $data & chr(10))
5 endfunc
6
7 LogWrite("Hello world!")
```

После выполнения этого скрипта в одной папке с ним будет создан файл `debug.log`, содержащий строку “Hello world!”. Функция `LogWrite` является

обёрткой над AutoIt вызовом `FileWrite`. Она будет удобна, если вам понадобиться отключить вывод в **лог-файл**. Для этого достаточно будет закомментировать в ней вызов `FileWrite`. Вы можете изменить путь до лог-файла и его имя с помощью константы `LogFile`.

Всегда предусматривайте способ **отладки** (обнаружения и устранения ошибок) вашего приложения. Самый простой подход заключается в печати на консоль или в файл наиболее важных решений, принятых его алгоритмом.

Первое условие, которое бот должен проверить, – это результат выбора цели. Попробуйте несколько раз выделить монстров с помощью мыши. Заметили ли вы элемент интерфейса, который отличается при наличии и отсутствии цели? Я имею в виду окно цели. Оно появляется каждый раз при выборе цели и пропадает при её убийстве или отмене по клавише Esc. Наш бот может найти это окно на экране с помощью функции `FFBestSpot` библиотеки FastFind.

Чтобы отличить окно цели от остальных, нам нужно выбрать уникальный для него цвет. Другими словами, надо найти такой цвет, который встречается только в окне цели. Для этого подошёл бы красный цвет полосы HP монстра. Код из листинга 2-24 проверяет, есть ли окно цели на экране.

Листинг 2-24. Функция `IsTargetExist`

```
func IsTargetExist()
    const $SizeSearch = 80
    const $MinNbPixel = 3
    const $OptNbPixel = 10
    const $PosX = 688
    const $PosY = 67

    $coords = FFBestSpot($SizeSearch, $MinNbPixel, $OptNbPixel, _
                         $PosX, $PosY, 0x871D18, 10)

    const $MaxX = 800
    const $MinX = 575
    const $MaxY = 100

    if not @error then
        if $MinX < $coords[0] and $coords[0] < $MaxX _
           and $coords[1] < $MaxY then

            LogWrite("IsTargetExist() - Success, coords = " & _
                     $coords[0] & ", " & $coords[1] & " pixels = " & _
                     $coords[2])
            return True
```

```
        else
            LogWrite("IsTargetExist() - Fail #1")
            return False
        endif
    else
        LogWrite("IsTargetExist() - Fail #2")
        return False
    endif
endfunc
```

Рассмотрим функцию `IsTargetExist` подробнее. Константы PosX и PosY – это примерные координаты полосы HP цели. Мы передаём их и красный цвет полосы (равный 871D18) в функцию `FFBestSpot` в качестве входных параметров. Она ищет указанную область по всему экрану.

Внимательный читатель заметит, что вместо окна цели может быть найдено окно состояния персонажа. Ведь в нём тоже встречается красный цвет на полоске HP персонажа. В таком случае бот всегда будет делать вывод, что цель есть. Чтобы избежать этой ошибки, мы проверяем координаты области, найденной функцией `FFBestSpot`. Сравниваем их (`coords[0]` и `coords[1]`) с максимальными (`MaxX` и `MaxY`) и минимальными (`MinX`) допустимыми значениями. Эти значения задают область экрана, в которой ожидается появление окна цели. Они зависят от разрешения экрана и конфигурации интерфейса игры. Поэтому вам придётся подбирать их самостоятельно.

В каждой ветви операторов `if` мы вызываем функцию `LogWrite`, чтобы отследить принятые решения. Благодаря этому мы сможем обнаружить возможные ошибки, связанные с несоответствием входных и выходных данных функции `IsTargetExist`.

`IsTargetExist` позволяет нам решить сразу две задачи:

1. Проверка успешности выбора цели в функции `SelectTarget`.
2. Проверка состояния атакуемого ботом монстра (жив или нет).

Скрипт `AnalysisBot.au3`, представленный в листинге 2-25, использует функцию `IsTargetExist` для проверки наличия цели.

Листинг 2-25. Скрипт `AnalysisBot.au3`

```
#include "FastFind.au3"

#RequireAdmin

Sleep(2000)

```

```
global const $LogFile = "debug.log"

func LogWrite($data)
    FileWrite($LogFile, $data & chr(10))
endfunc

func IsTargetExist()
    ; Смотрите реализацию в листинге 2-24
endfunc

func SelectTarget()
    LogWrite("SelectTarget()")
    while not IsTargetExist()
        Send("{F9}")
        Sleep(200)
    wend
endfunc

func Attack()
    LogWrite("Attack()")
    while IsTargetExist()
        Send("{F1}")
        Sleep(1000)
    wend
endfunc

func Pickup()
    Send("{F8}")
    Sleep(1000)
endfunc

while True
    SelectTarget()
    Attack()
    Pickup()
wend
```

Обратите внимание на новую реализацию функций `SelectTarget` и `Attack`. В `SelectTarget` бот пытается выделить цель в цикле до тех пор, пока функция `IsTargetExist` не вернёт значение `True`. Только после этого он переходит в функцию `Attack`. В ней бот продолжает атаковать монстра (выбирая действие “атака” по клавише F1) до тех пор, пока тот жив.

Мы печатаем в лог-файл названия функций `SelectTarget` и `Attack`, когда они получают управление. Этот вывод позволяет определить, которая из них вызывает `IsTargetExist`.

Дальнейшие улучшения

Теперь наш кликер выбирает действие, согласно игровой ситуации. Тем не менее, по-прежнему возможны случаи, когда бот допустит критическую ошибку и умрёт.

Первая проблема заключается в агрессивных монстрах. Большинство из них неагрессивны. Они остаются в одной и той же области карты, не реагируя на приближение игрока. Но некоторые из них в такой ситуации атакуют и преследуют.

Наш бот выбирает цель для атаки и бежит к ней. При этом он игнорирует всех других существ, которые встретятся ему по пути. Можно сказать, что они невидимы для бота, поскольку его алгоритм их не учитывает. Таким образом, агрессивные монстры могут напасть на бота, бегущего к своей цели. Он будет отрабатывать алгоритм сражения с одним противником, но на самом деле их может оказаться два или больше. Вместе они легко убьют бота.

Чтобы решить эту проблему, воспользуемся командой “выбор ближайшей цели”. На нашей панели горячих клавиш она доступна по нажатию F10. Ближайшая цель находится на минимальной (по сравнению с другими) дистанции от бота. Это значительно уменьшит время бота в пути, а значит и вероятность встречи с агрессивными монстрами.

Листинг 2-26 демонстрирует дополненную версию функции `SelectTarget`.

Листинг 2-26. Функция `SelectTarget`

```
func SelectTarget()
    LogWrite("SelectTarget()")

    while not IsTargetExist()
        Send("{F10}")
        Sleep(200)

        if IsTargetExist() then
            exitloop
        endif

        Send("{F9}")
        Sleep(200)
    wend
endfunc
```

Теперь бот в первую очередь пытается найти ближайшую цель по клавише F10. Только тогда, когда ему это не удалось, он использует команду `/target`. Таким образом бот всегда стремится выбрать ближайшего к нему монстра. Если тот окажется агрессивным, то побежит навстречу и будет ближе всего.

Вторую серьёзную проблему для бота представляют собой преграды на карте. При движении к цели он может зацепиться за камень или дерево и

застрять. Самое простое решение заключается в тайм-ауте на атаку. Если отведённое время на убийство монстра прошло, а цель осталась жива, можно предположить, что бот застрял. Тогда для обхода препятствия ему помогут случайные перемещения. Новые версии функций `Move` и `Attack` из листинга 2-27 демонстрируют это решение.

Листинг 2-27. Функции `Move` и `Attack`

```
func Move()
    SRandom(@MSEC)
    MouseClick("left", Random(300, 800), Random(170, 550), 1)
endfunc

func Attack()
    LogWrite("Attack()")

    const $TimeoutMax = 10
    $timeout = 0
    while IsTargetExist() and $timeout < $TimeoutMax
        Send("{F1}")
        Sleep(2000)

        Send("{F2}")
        Sleep(2000)

        $timeout += 1
    wend

    if $timeout == $TimeoutMax then
        Move()
    endif
endfunc
```

Мы добавили счётчик `timeout` в функцию `Attack`. На каждой итерации цикла `while` он **инкрементируется** и сравнивается с пороговым значением константы `TimeoutMax`. Когда счётчик достигает `TimeoutMax`, бот делает вывод, что застрял. В этом случае вызывается функция `Move`, которая симулирует щелчок левой кнопки мыши по точке со случайной координатой. Чтобы получить случайное число, используются функции AutoIt `SRandom` и `Random`. Первая из них инициализирует **генератор псевдослучайных чисел**. Вторая возвращает следующее число из очереди сгенерированных. В качестве параметров функция `Random` принимает границы интервала для случайного числа.

Возможно, вы заметили дополнительное действие, появившееся в новой функции `Attack`. Это симуляция нажатия клавиши F2. Мы можем назначить на неё любое атакующее умение персонажа, и бот будет применять его в сражении. Благодаря этому он сможет быстрее убивать монстров.

Теперь наш кликер способен самостоятельно работать достаточно долгое время. Он умеет обходить препятствия и первым атаковать агрессивных монстров. Но есть одно улучшение, способное значительно увеличить выживаемость бота. Речь идёт об использовании зелья восстановления здоровья, которое привязано к горячей клавише F5. Чтобы правильно его применять, необходимо анализировать полосу HP персонажа в окне состояния. Вы можете реализовать этот механизм самостоятельно в качестве упражнения. Алгоритм чтения уровня HP будет похож на функцию `IsTargetExist`.

Выводы

Мы реализовали кликера для игры Lineage 2. Он использует самые распространённые техники симуляции действий и анализа окна игрового приложения. Попробуем оценить их эффективность и обобщить результат на всех ботов этого типа.

Преимущества кликеров:

1. Простота разработки, отладки и расширения функциональности.
2. Просто адаптировать под любую версию игры, даже если её интерфейс поменялся.
3. Защититься от этого типа ботов достаточно сложно.

Недостатки кликеров:

1. Каждому пользователю приходится подгонять цвета и координаты искомых пикселей под своё разрешение экрана.
2. Бот может зависнуть в некоторых непредвиденных случаях (смерть персонажа, отключение от сервера и т.д.).
3. Тайм-ауты на симулируемые действия часто приводят к потере времени и низкой эффективности.
4. При анализе изображений на экране возможны ошибки. Поэтому в некоторых случаях бот будет выбирать неподходящие действия.

Кликеры хорошо подходят для автоматизации задач, состоящих из строгой последовательности шагов с минимальным количеством условий. Также обязательным требованием для их стабильной работы является относительно невысокая цена ошибки. То есть при выборе нескольких

неверных действия, бот должен иметь возможность вернуться в известное ему состояние.

Методы защиты от кликеров

Мы познакомились с основными принципами работы кликеров. Теперь рассмотрим этот тип ботов с точки зрения разработчика систем защиты. Как можно их обнаружить и помешать симулировать действия игрока? На этот вопрос мы найдём ответ в этом разделе.

В первой главе мы рассмотрели архитектуру типичной онлайн-игры. Как вы помните, её приложение состоит из двух частей: клиентской и серверной. Зачастую, система защиты придерживается такой же архитектуры и разделена на две части. Клиентская часть контролирует точки перехвата и внедрения данных на стороне пользователя (драйвера, ОС, приложение). Серверная часть следит за взаимодействием игрового приложения и сервером. Большинство техник по обнаружению кликеров работают на клиентской стороне.

Главная цель любой системы защиты заключается в обнаружении факта несанкционированного чтения или модификации игровых данных. Другими словами, отличить действия человека и программы. Когда нарушение обнаружено, у системы защиты есть несколько вариантов реакции:

1. Уведомить администратора игрового сервера о подозрительных действиях игрока. Для этого достаточно сделать запись в лог-файл на стороне сервера.
2. Разорвать соединение между подозрительным пользователем и сервером.
3. Заблокировать игрока по IP-адресу. Это предотвратит его дальнейшие попытки подключения к серверу.

Мы рассмотрим только алгоритмы обнаружения ботов, но не способы блокировки их работы. Поскольку самыми надёжными мерами пресечения нарушений будут не технические приёмы, а административные действия (например, блокировка аккаунта или штрафное время ожидания подключения к серверу).

Тестовое приложение

Для тестирования алгоритмов обнаружения ботов, мы воспользуемся приложением Notepad. Предположим, что это игровой клиент, который мы должны защитить. Напишем простейший AutoIt-скрипт, который выполняет роль кликера и вводит текст в Notepad. Тогда наша цель заключается в его обнаружении.

Листинг 2-28 демонстрирует скрипт `SimpleBot.au3`, который печатает буквы “a”, “b”, “c” в окне Notepad.

Листинг 2-28. Скрипт `SimpleBot.au3`

```
$hWnd = WinGetHandle("[CLASS:Notepad]")
WinActivate($hWnd)

Sleep(200)

while true
    Send("a")
    Sleep(1000)
    Send("b")
    Sleep(2000)
    Send("c")
    Sleep(1500)
wend
```

Для тестирования запустите Notepad, а затем скрипт `SimpleBot.au3`. Он переключится на нужное окно и будет вводить буквы в бесконечном цикле.

Скрипт `SimpleBot.au3` служит отправной точкой нашего исследования. Его цель в том, чтобы отличить симулируемые ботом нажатия клавиш от действий пользователя в окне Notepad. Прототипы алгоритмов защиты мы будем писать на AutoIt. Благодаря этому получится простой и компактный код для изучения. В реальных системах защиты предпочтительнее использовать компилируемые языки вроде C или C++.

Анализ действий игрока

Вычисление временных задержек

Скрипт `SimpleBot.au3` симулирует одни и те же действия в цикле. Их систематичность – это первое, что бросается в глаза при анализе работы бота. Ещё раз обратимся к его коду. Между каждым действием и предыдущим стоят строго определённые задержки. Человек не может действовать в таких точных временных интервалах. Более того, такие чёткие остановки не имеет никакого смысла в компьютерной игре, потому что зачастую пользователь должен реагировать на различные случайные

ситуации. Если кто-то ведёт себя подобным образом, очень вероятно что это программа.

Алгоритм защиты может замерять задержки между двумя одинаковыми действиями. Если они повторяются через одни и те же интервалы времени с разницей не более 100 миллисекунд, их наверняка выполняет бот. Попробуем реализовать такую защиту.

Скорость реакции среднестатистического человека равна примерно 300 миллисекундам. У профессиональных игроков она меньше и составляет порядка 150 миллисекунд.

Наш скрипт защиты должен выполнять две задачи: перехватывать действия пользователя и измерять временные задержки между ними. Код в листинге 2-29 реализует перехват нажатия клавиш.

Листинг 2-29. Перехват нажатия клавиш

```
global const $gKeyHandler = "_KeyHandler"

func _KeyHandler()
    $keyPressed = @HotKeyPressed

    LogWrite("_KeyHandler() - asc = " & asc($keyPressed) & " key = " & $keyPressed)
    AnalyzeKey($keyPressed)

    HotKeySet($keyPressed)
    Send($keyPressed)
    HotKeySet($keyPressed, $gKeyHandler)
endfunc

func InitKeyHooks($handler)
    for $i = 0 to 255
        HotKeySet(Chr($i), $handler)
    next
endfunc

InitKeyHooks($gKeyHandler)

while true
    Sleep(10)
wend
```

Мы применили функцию AutoIt `HotKeySet`, чтобы назначить **обработчик** (handler или hook) для нажатий клавиш. Она принимает на вход два параметра: код перехватываемой клавиши и ссылку на функцию-обработчик. Чтобы пройти по всем кодам от 0 до 255, в пользовательской функции `InitKeyHooks` используется цикл `for`. Обработчик `_KeyHandler`

назначается для всех клавиш. Алгоритм его работы выглядит следующим образом:

1. Вызвать функцию `AnalyzeKey` и передать ей код нажатой клавиши. Этот код хранится в макросе `@HotKeyPressed`.
2. Выключить перехват следующего нажатия обрабатываемой клавиши. Для этого снова вызывается функция `HotKeySet`. Данный шаг нужен, чтобы последующее нажатие обработало приложение Notepad, а не наш скрипт.
3. Вызвать функцию `Send` для симуляции нажатия обрабатываемой клавиши в Notepad. Этот шаг нужен, поскольку нажатие пользователя получил скрипт, а не Notepad.
4. Включить перехват скриптом последующих нажатий с помощью `HotKeySet`.

Листинг 2-30 демонстрирует код функции `AnalyzeKey`.

Листинг 2-30. Функция `AnalyzeKey`

```
global $gTimeSpanA = -1
global $gPrevTimestampA = -1

func AnalyzeKey($key)
    local $timestamp = (@SEC * 1000 + @MSEC)
    LogWrite("AnalyzeKey() - key = " & $key & " msec = " & $timestamp)
    if $key <> 'a' then
        return
    endif

    if $gPrevTimestampA = -1 then
        $gPrevTimestampA = $timestamp
        return
    endif

    local $newTimeSpan = $timestamp - $gPrevTimestampA
    $gPrevTimestampA = $timestamp

    if $gTimeSpanA = -1 then
        $gTimeSpanA = $newTimeSpan
        return
    endif

    if Abs($gTimeSpanA - $newTimeSpan) < 100 then
        MsgBox(0, "Alert", "Обнаружен бот-кликер!")
    endif
endfunc
```

В функции `AnalyzeKey` мы измеряем задержки между нажатиями клавиши “a”. Две глобальные переменные хранят текущее состояние алгоритма:

1. `gPrevTimestampA` – это момент времени (timestamp) первого нажатия.

2. `gTimeSpanA` – это задержка между первым и вторым нажатиями.

При старте скрипта обоим переменным присваивается значение -1, которое соответствует неинициализированному состоянию. Нашему алгоритму требуется перехватить как минимум три нажатия клавиш, чтобы обнаружить бота. Первое нажатие инициализирует переменную `gPrevTimestampA`:

```
    if $gPrevTimestampA = -1 then
        $gPrevTimestampA = $timestamp
        return
    endif
```

Момент времени второго нажатия мы используем для расчёта переменной `gTimeSpanA`. Она равна разности между временем первого и второго нажатий:

```
    local $newTimeSpan = $timestamp - $gPrevTimestampA
    $gPrevTimestampA = $timestamp

    if $gTimeSpanA = -1 then
        $gTimeSpanA = $newTimeSpan
        return
    endif
```

После третьего нажатия мы можем вычислить задержку второй раз (переменная `newTimeSpan`) и сравнить её с предыдущей (значение `gTimeSpanA`):

```
    if Abs($gTimeSpanA - $newTimeSpan) < 100 then
        MsgBox(0, "Alert", "Clicker bot detected!")
    endif
```

Если разница между первой и второй задержкой меньше 100 миллисекунд, алгоритм защиты выводит сообщение об обнаружении бота.

Полный код защиты представлен в скрипте `TimeSpanProtection.au3` из листинга 2-31. В нём мы опустили реализацию функций `_KeyHandler` и `AnalyzeKey`, поскольку рассмотрели их ранее.

Листинг 2-31. Скрипт `TimeSpanProtection.au3`

```
global const $gKeyHandler = "_KeyHandler"
global const $kLogFile = "debug.log"

global $gTimeSpanA = -1
global $gPrevTimestampA = -1

func LogWrite($data)
```

```
    FileWrite($kLogFile, $data & chr(10))
endfunc

func _KeyHandler()
    ; См листинг 2-29
endfunc

func InitKeyHooks($handler)
    for $i = 0 to 256
        HotKeySet(Chr($i), $handler)
    next
endfunc

func AnalyzeKey($key)
    ; См листинг 2-30
endfunc

InitKeyHooks($gKeyHandler)

while true
    Sleep(10)
wend
```

Анализ последовательности действий

Мы можем незначительно изменить скрипт `SimpleBot.au3`, чтобы обойти защиту `TimeSpanProtection.au3`. Для этого заменим фиксированные задержки между действиями на случайные. Листинг 2-32 демонстрирует исправленную версию бота.

Листинг 2-32. Скрипт `RandomDelayBot.au3`

```
SRandom(@MSEC)

$hWnd = WinGetHandle("[CLASS:Notepad]")
WinActivate($hWnd)

Sleep(200)

while true
    Send("a")
    Sleep(Random(800, 1200))
    Send("b")
    Sleep(Random(1700, 2300))
    Send("c")
    Sleep(Random(1300, 1700))
wend
```

Каждый раз, в вызов `Sleep` мы передаём случайное число, полученное из функции `Random`. Попробуйте протестировать нового бота вместе с защитой `TimeSpanProtection.au3`. Теперь она не обнаружит кликера. Можем ли мы её улучшить?

У скрипта `RandomDelayBot.au3` по-прежнему есть закономерность, которая сразу видна человеку, следящему за его работой. Речь идёт о

последовательности нажимаемых кнопок. Очевидно, что игрок не способен безошибочно повторять свои действия десятки и сотни раз. Даже если он и захочет это сделать, в какой-то момент он ошибётся и нажмёт не ту клавишу.

Перепишем скрипт защиты так, чтобы вместо временных задержек он анализировал последовательность нажатий клавиш. Для этого надо изменить функцию `AnalyzeKey`, как показано в листинге 2-33.

Листинг 2-33. Функция `AnalyzeKey`

```
global const $gActionTemplate[3] = ['a', 'b', 'c']
global $gActionIndex = 0
global $gCounter = 0

func Reset()
    $gActionIndex = 0
    $gCounter = 0
endfunc

func AnalyzeKey($key)
    LogWrite("AnalyzeKey() - key = " & $key);

    $indexMax = UBound($gActionTemplate) - 1
    if $gActionIndex <= $indexMax and $key <> $gActionTemplate[$gActionIndex] then
        Reset()
        return
    endif

    if $gActionIndex < $indexMax and $key = $gActionTemplate[$gActionIndex] then
        $gActionIndex += 1
        return
    endif

    if $gActionIndex = $indexMax and $key = $gActionTemplate[$gActionIndex] then
        $gCounter += 1
        $gActionIndex = 0

        if $gCounter = 3 then
            MsgBox(0, "Alert", "Обнаружен бот-кликер!")
            Reset()
        endif
    endif
endfunc
```

Новый вариант функции `AnalyzeKey` использует глобальную константу и две переменные:

1. `gActionTemplate` – это массив с последовательностью действий, которую выполняет предполагаемый бот.
2. `gActionIndex` – индекс массива `gActionTemplate`, который соответствует последнему перехваченному нажатию.

3. `gCounter` – число обнаруженных повторений последовательности действий.

В функции `AnalyzeKey` есть три основных условия для обработки нажатия клавиши. Первое из них выполняется, если нажатие не соответствует ни одному элементу массива `gActionTemplate`:

```
    $indexMax = UBound($gActionTemplate) - 1
    if $gActionIndex <= $indexMax and $key <> $gActionTemplate[$gActionIndex] then
        Reset()
        return
    endif
```

В этом случае мы вызываем функцию `Reset`, которая сбрасывает в ноль значения переменных `gActionIndex` и `gCounter`. После этого мы выходим из `AnalyzeKey`.

Второе условие обработки нажатия выполняется, когда перехваченное действие встречается в массиве `gActionTemplate`, кроме того этот элемент не последний и его индекс равен `gActionIndex`:

```
    if $gActionIndex < $indexMax and $key = $gActionTemplate[$gActionIndex] then
        $gActionIndex += 1
        return
    endif
```

Выполнение условия означает, что нажатие попадает в предполагаемую последовательность действий бота. В этом случае мы инкрементируем переменную `gActionIndex` и ожидаем новое нажатие, чтобы сравнить его со следующим элементом последовательности.

Третье условие выполняется, когда перехваченное нажатие соответствует последнему элементу массива `gActionTemplate`:

```
if $gActionIndex = $indexMax and $key = $gActionTemplate[$gActionIndex]
then
        $gCounter += 1
        $gActionIndex = 0
        if $gCounter = 3 then
            MsgBox(0, "Alert", "Clicker bot detected!")
            Reset()
        endif
    endif
```

В этом случае мы инкрементируем счётчик совпадения последовательностей `gCounter` и сбрасываем значение `gActionIndex`.

Таким образом мы готовы к обнаружению следующих действий бота.

Если ожидаемая последовательность действий происходит три раза подряд, скрипт делает вывод, что её симулирует бот. Тогда пользователю выдаётся соответствующее сообщение. В этом случае счётчик `gCounter` сбрасывается в ноль и алгоритм защиты начинает свою работу сначала.

Вы можете запустить скрипты `ActionSequenceProtection.au3` и `RandomDelayBot.au3` для тестирования защиты. Теперь бот будет обнаружен.

Очевидно, что рассмотренный алгоритм может ошибиться. Он примет игрока за бота, если тот трижды повторит одни и те же действия. Вероятность такой ошибки можно уменьшить, если мы увеличим пороговое значение для счётчика `$gCounter` в следующем условии:

```
        if $gCounter = 3 then
            MsgBox(0, "Alert", "Clicker bot detected!")
            Reset()
        endif
```

К сожалению, у скрипта защиты `ActionSequenceProtection.au3` есть и другой серьёзный недостаток. Он способен обнаружить только бота, который запрограммирован на последовательность нажатий “a”, “b”, “c”. Если кликер вместо этого будет выполнять “a”, “c”, “b”, то алгоритм не сможет его обнаружить.

Изменим нашего бота согласно листингу 2-34. Это позволит ему обойти защиту `ActionSequenceProtection.au3`.

Листинг 2-34. Скрипт `RandomActionBot.au3`

```
SRandom(@MSEC)

$hWnd = WinGetHandle("[CLASS:Notepad]")
WinActivate($hWnd)

Sleep(200)

while true
    Send("a")
    Sleep(1000)

    if Random(0, 9, 1) < 5 then
        Send("b")
        Sleep(2000)
    endif

    Send("c")
```

```
    Sleep(1500)
wend
```

Теперь симулируемая ботом последовательность действий случайна. Он пропускает нажатие клавиши “b” после “a” с вероятностью порядка 50%. Это приводит к тому, что условия функции `AnalyzeKey` на обнаружение бота перестают выполняться. Каждый раз, когда бот пропускает “b”, алгоритм защиты сбрасывает счётчик `gCounter` в ноль. Таким образом, он никогда не достигает порогового значения.

Мы можем обнаружить бота `RandomActionBot.au3`, если немного изменим защитный алгоритм. Вместо проверки нажатий клавиш “на лету”, он должен записывать их в один большой файл. Когда этот файл достигнет максимально допустимого размера, скрипт должен его прочитать и проверить на наличие часто повторяющихся последовательностей действий. Если они встречаются, это может быть сигналом о том, что их выполняет программа. В случае бота `RandomActionBot.au3`, такими последовательностями будут:

1. “a”, “c”.
2. “a”, “b”, “c”.

Сканирование процессов

Есть принципиально иной подход к обнаружению кликеров. Вместо того, чтобы анализировать действия игрока, алгоритм защиты может найти бота в списке запущенных процессов ОС.

Скрипт `ProcessScanProtection.au3`, приведённый в листинге 2-35, демонстрирует этот подход.

Листинг 2-35. Скрипт `ProcessScanProtection.au3`

```
global const $kLogFile = "debug.log"

func LogWrite($data)
    FileWrite($kLogFile, $data & chr(10))
endfunc

func ScanProcess($name)
    local $processList = ProcessList($name)

    if $processList[0][0] > 0 then
        LogWrite("Name: " & $processList[1][0] & " PID: " & $processList[1][1])
        MsgBox(0, "Alert", "Обнаружен бот-кликер!")
    endif
endfunc

```

```
16 while true
17     ScanProcess("AutoHotKey.exe")
18     Sleep(5000)
19 wend
```

Мы можем получить список запущенных в данный момент процессов с помощью AutoIt функции `ProcessList`. У неё есть единственный необязательный параметр: имя процесса, который нужно найти. Если его передать, функция вернёт список из одного элемента в случае успешного поиска. Предположим, что защита ищет процесс **интерпретатора** `AutoHotKey.exe`, который выполняет скрипт бота. `ProcessList` возвращает двумерный массив, представленный в таблице 2-8.

Таблица 2-8. Элементы массива, возвращаемого `ProcessList`

Элемент массив	Описание
`processList[0][0]`	Количество найденных процессов.
`processList[1][0]`	Имя первого процесса в списке.
`processList[1][1]`	Идентификатор (ID или PID) первого процесса в списке.

Если элемент массива `processList[0][0]` не равен нулю, процесс `AutoHotKey.exe` в данный момент запущен и работает.

Почему мы ищем процесс `AutoHotKey.exe`, а не `AutoIt.exe`? Дело в том, что мы не сможем протестировать скрипт `ProcessScanProtection.au3` на нашем тестовом боте `SimpleBot.au3`. Оба скрипта написаны на языке AutoIt. Это значит, что как только мы щёлкнем два раза мышью по иконке `ProcessScanProtection.au3`, ОС запустит процесс интерпретатора `AutoIt.exe`, который и выполнит наш скрипт. Из-за этого алгоритм защиты будет всегда находить процесс `AutoIt.exe`, независимо от того запущен бот или нет.

Перепишем нашего тестового бота на языке AutoHotKey. Результат приведён в листинге 2-36.

Листинг 2-36. Скрипт `SimpleBot.ahk`

```
1 WinActivate, Untitled - Notepad
2 Sleep, 200
3
4 while true
```

```
{
    Send, a
    Sleep, 1000
    Send, b
    Sleep, 2000
    Send, c
    Sleep, 1500
}
```

Вы можете сравнить скрипты `SimpleBot.ahk` и `SimpleBot.au3`. Они выглядят похоже. Единственное отличие заключается в синтаксисе вызова функций. В AutoHotKey параметры указываются не в скобках, а через запятую и пробел после имени функции.

Теперь мы можем протестировать скрипт защиты `ProcessScanProtection.au3`. Для этого выполните следующие шаги:

1. Запустите приложение Notepad.
2. Запустите скрипт `ProcessScanProtection.au3`.
3. Запустите тестового бота `SimpleBot.ahk`. Не забудьте перед этим установить на свой компьютер интерпретатор AutoHotKey.
4. Ожидайте, пока алгоритм защиты не обнаружит бота. Когда это случится, откроется диалоговое окно с сообщением.

Есть несколько методов для обхода такого типа защиты. Самый простой из них заключается в применении компилятора скриптов. Компилятор собирает скрипт и интерпретатор `AutoHotKey.exe` в единый исполняемый EXE-файл. Имя этого файла будет соответствовать имени процесса в списке, возвращаемом функцией `ProcessList`. Таким образом, алгоритм защиты `ProcessScanProtection.au3` не сработает.

Для компилирования скрипта `SimpleBot.ahk` выполните следующие шаги:

1. Запустите приложение компилятора AutoHotKey. Его окно выглядит как на иллюстрации 2-14. Путь к нему по умолчанию:

```
C:\Program Files (x86)\AutoHotkey\Compiler\Ahk2Exe.exe
```

2. Выберите скрипт `SimpleBot.ahk` в качестве исходного файла. Диалоговое окно для выбора файла открывается по кнопке “Browse” напротив текста “Source (script file)”.
3. Не указывайте имя выходного файла в поле “Destination (.exe file)”. В этом случае в той же папке, где находится скрипт, будет создан EXE-

файл с таким же именем.
4. Нажмите кнопку “> Convert <”. После окончания процесса компиляции вы увидите сообщение.

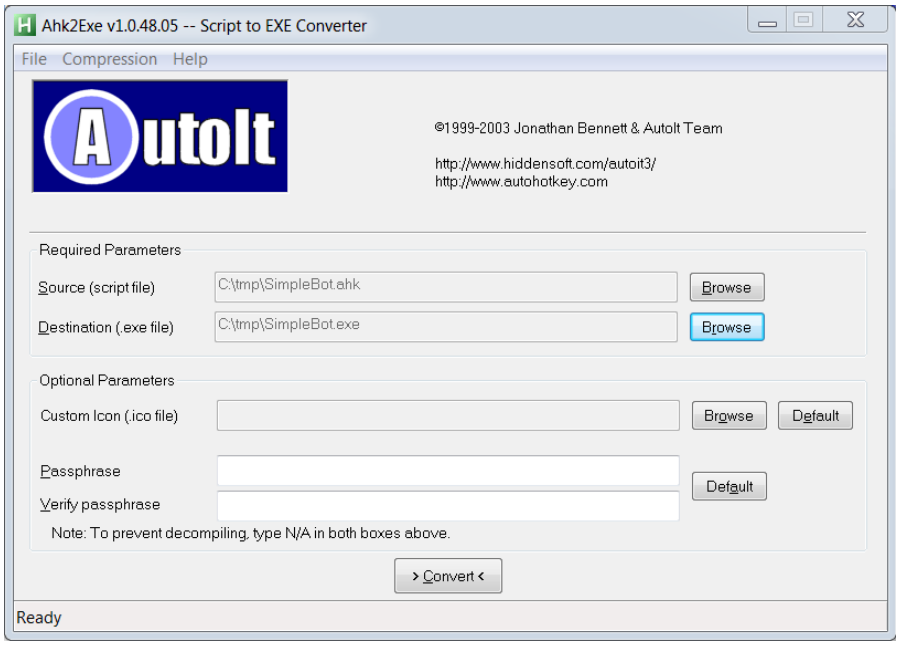


Иллюстрация 2-14. Окно компилятора AutoHotKey

Попробуйте запустить полученный EXE-файл с именем `SimpleBot.exe`. Он ведёт себя точно так же, как и скрипт `SimpleBot.ahk`. Единственное отличие в том, что алгоритм защиты `ProcessScanProtection.au3` не может его обнаружить. Это происходит из-за того, что процесс бота теперь называется `SimpleBot.exe`, а не `AutoHotKey.exe`.

Вычисление хэш-суммы запускаемого файла

Можем ли мы усовершенствовать скрипт `ProcessScanProtection.au3` так, чтобы он обнаруживал скомпилированную версию бота `SimpleBot.exe`? Как мы выяснили, имя исполняемого файла легко поменять в отличие от его содержания. EXE-файл представляет собой машинный код, который выполняется процессором, и заголовки с метаинформацией. Неверное изменение любой из этих частей приведёт к ошибке приложения при старте.

Если система защиты будет искать бота не по имени процесса, а по содержанию его исполняемого файла, её будет сложнее обойти. Вот

несколько идей того, что может проверять алгоритм такой защиты:

1. Рассчитывать **хэш-суммы** исполняемых файлов всех запущенных процессов и сравнивать их с предопределёнными значениями.
2. Проверять последовательность байт в определённом месте каждого из этих исполняемых файлов.
3. Искать заданную последовательность байт по всему файлу.

Попробуем реализовать первый подход. Скрипт `Md5ScanProtection.au3` из листинга 2-37 считает хэш-сумму по алгоритму MD5 для исполняемого файла каждого из запущенных процессов. Если она совпала с искомой, алгоритм делает вывод о наличии работающего бота.

Листинг 2-37. Скрипт `Md5ScanProtection.au3`

```
#include <Crypt.au3>

global const $kLogFile = "debug.log"
global const $kCheckMd5[2] = ["0x3E4539E7A04472610D68B32D31BF714B", _
                              "0xD960F13A44D3BD8F262DF625F5705A63"]

func LogWrite($data)
    FileWrite($kLogFile, $data & chr(10))
endfunc

func _ProcessGetLocation($pid)
    local $proc = DllCall('kernel32.dll', 'hwnd', 'OpenProcess', 'int', _
                          BitOR(0x0400, 0x0010), 'int', 0, 'int', $pid)

    if $proc[0] = 0 then
        return ""
    endif

    local $struct = DllStructCreate('int[1024]')
    DllCall('psapi.dll', 'int', 'EnumProcessModules', 'hwnd', $proc[0], _
            'ptr', DllStructGetPtr($struct), 'int', _
            DllStructGetSize($struct), 'int_ptr', 0)

    local $return = DllCall('psapi.dll', 'int', 'GetModuleFileNameEx', _
        'hwnd', $proc[0], 'int', DllStructGetData($struct, 1), _
        'str', '', 'int', 2048)

    if StringLen($return[3]) = 0 then
        return ""
    endif

    return $return[3]
endfunc

func ScanProcess()
    local $processList = ProcessList()

    for $i = 1 to $processList[0][0]
        local $path = _ProcessGetLocation($processList[$i][1])
        local $md5 = _Crypt_HashFile($path, $CALG_MD5)
        LogWrite("Name: " & $processList[$i][0] & " PID: " & _
                 $processList[$i][1] & " Path: " & $path & " md5: " & $md5)

```

```
        for $j = 0 to Ubound($kCheckMd5) - 1
            if $md5 == $kCheckMd5[$j] then
                MsgBox(0, "Alert", "Обнаружен бот-кликер!")
            endif
        next
    next
endfunc

while true
    ScanProcess()
    Sleep(5000)
wend
```

Рассмотрим скрипт `Md5ScanProtection.au3` подробнее. Весь алгоритм обнаружения бота реализован в функции `ScanProcess`, которая вызывается в цикле `while` каждые пять секунд. В ней читается список запущенных процессов с помощью AutoIt-вызова `ProcessList`. Его результат сохраняется в переменную `processList`. После этого цикл `for` проходит по полученному списку. Для каждого его элемента функция `_ProcessGetLocation` читает путь к исполняемому файлу, машинный код которого был загружен в память процесса. Полученный путь передаётся в AutoIt-функцию `_Crypt_HashFile`, которая считает хэш-сумму по содержимому всего файла. На заключительном шаге алгоритма происходит сравнение рассчитанной хэш-суммы с искомыми значениями из глобального массива `kCheckMd5`. В нашем примере этот массив содержит MD5-суммы файлов `SimpleBot.exe` и `AutoHotKey.exe`.

Рассмотрим функцию `_ProcessGetLocation`. В ней происходит три WinAPI-вызова через AutoIt обёртку `DllCall`:

1. `OpenProcess`
2. `EnumProcessModules`
3. `GetModuleFileNameEx`

Первый вызов `OpenProcess` возвращает дескриптор процесса по его идентификатору. С помощью дескриптора можно запросить дополнительную информацию о процессе через WinAPI.

Следующая функция `EnumProcessModules` читает список **модулей** процесса в массив `struct`. Обычно процесс состоит из нескольких модулей. Каждый из них содержит машинный код исполняемого файла или динамической библиотеки DLL. Этот код загружается в память процесса при старте. Первый модуль в списке всегда соответствует исполняемому файлу. Его мы и передаём в функцию `GetModuleFileNameEx`.

Она извлекает из метаинформации модуля путь к соответствующему ему файлу.

Попробуйте запустить скрипт `Md5ScanProtection.au3` и оба варианта бота: `SimpleBot.ahk` и `SimpleBot.exe`. Новый алгоритм должен их обнаружить.

Может случиться так, что скрипт `SimpleBot.ahk` не будет обнаружен. Это означает, что ваша версия интерпретатора AutoHotKey отличается от моей. Чтобы это исправить, добавьте в массив `kCheckMd5` его хэш-сумму. Вы можете узнать её из лог-файла `debug.log` с отладочной информацией. В него пишутся все прочитанные защитой `Md5ScanProtection.au3` процессы и их MD5-суммы.

Есть несколько способов улучшить нашего бота, чтобы обойти алгоритм защиты `Md5ScanProtection.au3`. Все они связаны с изменением содержания исполняемого файла. Наиболее простые варианты следующие:

1. Сделать незначительное изменение в коде скрипта `SimpleBot.ahk` (например, уменьшить задержку на пару миллисекунд) и скомпилировать его по новой.
2. Изменить заголовок с метаинформацией исполняемого файла `AutoHotKey.exe`. Для этого можно воспользоваться редактором **HT Editor**.

Изменение машинного кода, записанного в исполняемый файл, чревато повреждением приложения. В этом случае оно завершится с ошибкой при старте. Но метаинформация, хранимая в **заголовке COFF** (Common Object File Format), не так чувствительна к изменениям. У заголовка есть несколько стандартных полей. Одно из них – время создания файла. Очевидно, изменение этого поля никак не повлияет на функциональность приложения. В то же время исправленное время создания файла приведёт к другому результату расчёта MD5-суммы. В результате алгоритм защиты `Md5ScanProtection.au3` не сможет обнаружить бота.

Выполните следующие шаги, чтобы изменить время создания файла в заголовке COFF:

1. Запустите приложение HT Editor с правами администратора. Для удобства скопируйте сначала исполняемый файл редактора в папку к

`AutoHotKey.exe`.

2. В окне редактора нажмите клавишу F3, чтобы вызвать диалог открытия файла (“open file”).
3. Нажмите клавишу Tab, чтобы перейти к списку файлов (“files”). Найдите в нём `AutoHotKey.exe` и нажмите Enter.
4. Нажмите F6, чтобы открыть диалог выбора режима редактирования (“select mode”). В нём включите режим “- pe/header”. После этого вы увидите список заголовков файла `AutoHotKey.exe`.
5. Выберите пункт “COFF header” и нажмите Enter. Перейдите на поле “time-data stamp” заголовка.
6. Нажмите F4 для редактирования поля и измените его. Иллюстрация 2-15 демонстрирует эту операцию.
7. Нажмите F4 и выберите вариант “Yes” в диалоге подтверждения сохранения изменений.

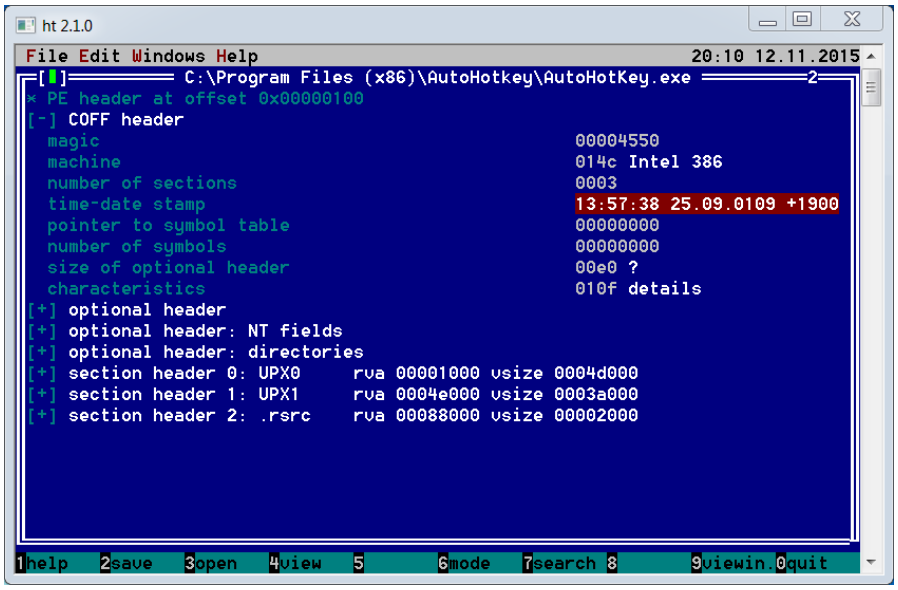


Иллюстрация 2-15. Изменение времени создания файла в HT editor

В результате мы получим файл `AutoHotKey.exe`, содержание которого отличается от исходного. Попробуйте запустить его и указать нашего бота `SimpleBot.ahk` в диалоге открытия скрипта. Алгоритм защиты `Md5ScanProtection.au3` не сможет его обнаружить.

Можно исправить алгоритм защиты так, чтобы он игнорировал все заголовки исполняемого файла при подсчёте хэш-суммы. Тогда разработчику бота придётся менять его машинный код. Альтернативное

решение для усиления защиты – считать MD5 не для всего содержания файла, а только для небольшого набора байтов из строго определённого места. В этом случае для обхода алгоритма надо будет точно знать это место.

Проверка состояния клавиатуры

Windows предоставляет механизм уровня ядра, который позволяет отличить реальное нажатие клавиши от симулируемого. Рассмотрим, как можно использовать этот механизм для обнаружения кликеров.

Прежде всего, мы должны перехватить событие нажатия клавиши на низком уровне. В этом нам поможет WinAPI-вызов `SetWindowsHookEx`. Принцип его работы похож на функцию AutoIt `HotKeySet`: он устанавливает обработчик для различных типов событий ОС. Первый входной параметр `SetWindowsHookEx` определяет этот тип. В нашем случае он должен быть равен `WH_KEYBOARD_LL`, что соответствует событиям клавиатуры.

Теперь мы должны реализовать функцию-обработчик события. Она получает входным параметром структуру типа `KBDLLHOOKSTRUCT`, которая содержит полную информацию о перехваченном событии. У этой структуры есть поле `flags`. Если в нём присутствует **флаг** (то есть бит в определённой позиции) `LLKHF_INJECTED`, перехваченное нажатие клавиши было симулировано WinAPI-функцией `SendInput` или `keybd_event`. Если флага `LLKHF_INJECTED` нет, источником события является клавиатура. Подменить поле `flags` структуры `KBDLLHOOKSTRUCT` достаточно сложно, поскольку оно выставляется на уровне ядра ОС.

Скрипт `KeyboardCheckProtection.au3` из листинга 2-38 демонстрирует проверку флага `LLKHF_INJECTED`.

Листинг 2-38. Скрипт `KeyboardCheckProtection.au3`

```
#include <WinAPI.au3>

global const $kLogFile = "debug.log"
global $gHook

func LogWrite($data)
    FileWrite($kLogFile, $data & chr(10))
endfunc

func _KeyHandler($nCode, $wParam, $lParam)
    if $nCode < 0 then
        return _WinAPI_CallNextHookEx($gHook, $nCode, $wParam, $lParam)
```

```
    endIf

    local $keyHooks = DllStructCreate($tagKBDLLHOOKSTRUCT, $lParam)

    LogWrite("_KeyHandler() - keyccode = " & DllStructGetData($keyHooks, "vkCode"));

    local $flags = DllStructGetData($keyHooks, "flags")
    if $flags = $LLKHF_INJECTED then
        MsgBox(0, "Alert", "Обнаружен бот-кликер!")
    endif

    return _WinAPI_CallNextHookEx($gHook, $nCode, $wParam, $lParam)
endfunc

func InitKeyHooks($handler)
    local $keyHandler = DllCallbackRegister($handler, "long", "int;wparam;lparam")
    local $hMod = _WinAPI_GetModuleHandle(0)
    $gHook = _WinAPI_SetWindowsHookEx($WH_KEYBOARD_LL, _
        DllCallbackGetPtr($keyHandler), $hMod)
endfunc

InitKeyHooks("_KeyHandler")

while true
    Sleep(10)
wend
```

Алгоритм назначения обработчика нажатий клавиш похож на тот, который мы применяли в скриптах `TimeSpanProtection.au3` и `ActionSequenceProtection.au3`. Только в данном случае мы делаем вызов WinAPI через AutoIt-обёртку `_WinAPI_SetWindowsHookEx` в функции `InitKeyHooks`. Таким образом мы инициализируем обработчик `_KeyHandler`, который будет перехватывать все события клавиатуры.

Функция `InitKeyHooks` выполняет следующие шаги:

1. Регистрирует обработчик `_KeyHandler` через AutoIt-функцию `DllCallbackRegister`. Это позволит передать его в вызовы WinAPI.
2. Читает в переменную `hMod` дескриптор первого модуля (нумерация начинается с нуля) текущего процесса через обёртку `_WinAPI_GetModuleHandle`. Не забудьте, что наш скрипт выполняется в интерпретаторе AutoIt.
3. Добавляет `_KeyHandler` в цепочку обработчиков через WinAPI-вызов `SetWindowsHookEx`. В неё мы должны передать дескриптор модуля, в котором этот обработчик реализован. В нашем случае это переменная `hMod`.

Алгоритм проверки флага `LLKHF_INJECTED` в обработчике `_KeyHandler` выглядит следующим образом:

1. Проверить значение параметра `nCode`. Если оно меньше нуля, мы передаём событие дальше по цепочке обработчиков. В этом случае оно не содержит нужной нам структуры `KBDLLHOOKSTRUCT`.
2. Если параметр `nCode` не равен нулю, вызвать функцию `DllStructCreate` и передать в неё `lParam`. Таким образом мы получаем структуру `KBDLLHOOKSTRUCT`.
3. Прочитать поле `flags` из `KBDLLHOOKSTRUCT` с помощью функции `DllStructGetData`.
4. Проверить наличие флага `LLKHF_INJECTED`. Если он присутствует, нажатие клавиши было симулировано ботом.

Для тестирования защиты `KeyboardCheckProtection.au3` запустите Notepad и бота `SimpleBot.au3`. Как только он выполнит первое нажатие клавиши, вы увидите сообщение об его обнаружении.

Есть несколько способов обойти подобную защиту. Для этого надо симулировать нажатия так, чтобы ядро ОС воспринимало их идущими от клавиатуры. Эти способы следующие:

1. Использовать виртуальную машину (virtual machine или VM).
2. Использовать специальный драйвер клавиатуры вместо WinAPI-функций `SendInput` и `keybd_event` для симуляции нажатий. Пример такого драйвера – InpOut32.
3. Эмулировать клавиатуру или мышь на специальном устройстве. Мы рассмотрим этот подход в пятой главе.

Самый простой в реализации вариант – использование виртуальной машины. У неё есть **виртуальные драйверы устройств**. Они решают две задачи: эмулируют устройства для **гостевой ОС** (запущенной внутри VM) и предоставляют доступ к реальным устройствам. Все события симулируемые на хост-системе (на которой запускается VM) и идущие от реальных устройств проходят через виртуальные драйверы. Из-за этого гостевая ОС не может отличить их источник. Поэтому симулируемые ботом нажатия не будут иметь флага `LLKHF_INJECTED`.

Для запуска VM и нашего тестового бота выполните шаги:

1. Установите одну из следующих виртуальных машин:
   * Virtual Box

    * VMWare Player
    * Windows Virtual PC
2. Установите Windows в качестве гостевой ОС.
3. Запустите в ней Notepad и скрипт `KeyboardCheckProtection.au3`.
4. Запустите скрипт `VirtualMachineBot.au3` на хост-системе.

Скрипт `VirtualMachineBot.au3` из листинга 2-39 представляет адаптированную версию нашего бота.

Листинг 2-39. Скрипт `VirtualMachineBot.au3`

```
Sleep(2000)

while true
    Send("a")
    Sleep(1000)
    Send("b")
    Sleep(2000)
    Send("c")
    Sleep(1500)
wend
```

Скрипт `VirtualMachineBot.au3` отличается от `SimpleBot.au3` процедурой переключения на окно Notepad. Теперь бот не может самостоятельно его найти, поскольку Notepad запущен на гостевой ОС. Мы добавили двухсекундную задержку после старта скрипта, чтобы у вас было время переключиться на окно VM и Notepad внутри неё. Алгоритм защиты `KeyboardCheckProtection.au3` не сможет обнаружить скрипт `VirtualMachineBot.au3`.

Выводы

Мы рассмотрели методы обнаружения кликеров. Каждый из них имеет свои достоинства и недостатки. Чтобы ваш бот смог обойти защиту, вы должны хорошо изучить её алгоритм. Следующие подходы помогут вам в этом:

1. Перехват выполняемых защитой WinAPI-вызовов. Для этой цели подойдёт приложение API Monitor.
2. Применение методов **реверс-инжиниринга** для изучения исполняемых файлов и DLL-библиотек системы защиты.
3. Тестирование различных механизмов симуляции нажатий клавиш. Это поможет выяснить, на что именно реагирует защита.

Современные системы защиты на стороне клиента совмещают в себе несколько алгоритмов обнаружения кликеров. Поэтому у хорошего бота должны быть средства их преодоления.

Внутриигровые боты

В этой главе мы рассмотрим внутриигровых ботов. Сначала познакомимся с инструментами для их разработки. Большая часть этих инструментов нужна для анализа игрового приложения, в которое должен встраиваться бот. Затем мы рассмотрим структуру памяти типичного процесса в ОС Windows. Научимся методам поиска, чтения и записи переменных в работающее игровое приложение. После этого разработаем простого бота для игры Diablo 2, чтобы закрепить полученные знания. В конце главы, мы рассмотрим алгоритмы защиты от внутриигровых ботов.

Инструменты для разработки

Разработка внутриигровых ботов происходит на более низком уровне по сравнению с кликерами. В ней приходится оперировать простыми абстракциями ОС. Поэтому наши инструменты будут сложнее, чем в прошлой главе.

Язык программирования

В этой главе мы будем использовать только язык C++. Для компиляции и работы с кодом рекомендую вам бесплатную IDE Microsoft Visual Studio вместо открытого набора инструментов MinGW. Проблема в том, что MinGW плохо интегрируется с некоторыми Windows библиотеками (например `dbghelp.dll`). Вы можете пробовать компилировать примеры этой главы с MinGW, но будьте готовы переключиться на Visual Studio IDE.

Чтобы запустить последнюю версию Visual Studio IDE, обновите браузер Microsoft Edge.

Для доступа к Windows Native API и линковки с системной библиотекой `ntdll.dll` вам понадобится Windows SDK.

Отладчики

Отладчик – это инструмент для тестирования и поиска ошибок в приложениях. Обычно им пользуются разработчики программ для исправления своего кода. Однако у отладчиков есть возможности, которые оказываются полезными для исследования чужих приложений.

Бесплатный отладчик **OllyDbg** мы будем активно использовать на протяжении всей главы. Простой и понятный интерфейс пользователя является его главным преимуществом. Также OllyDbg предоставляет широкие возможности для анализа

Windows приложений без исходного кода. Главный его недостаток заключается в поддержке только 32-битный приложений. Рекомендую вам использовать последнюю версию OllyDbg 2.0.

Отладчик с открытым исходным кодом **x64dbg** поддерживает и 32-битные, и 64-битные приложения. Некоторые возможности OllyDbg в нём отсутствуют, поэтому часть вычислений вам придётся делать самостоятельно. Я рекомендую использовать x64dbg только для отладки 64-битных приложений и OllyDbg в остальных случаях.

WinDbg – многоцелевой бесплатный отладчик для работы с пользовательскими приложениями, драйверами устройств, системными библиотеками и ядром ОС. Он предоставляет некоторые возможности недоступные в OllyDbg и x64dbg, а также поддерживает 32 и 64-битные приложения. Единственный серьёзный недостаток WinDbg заключается в неудобном пользовательском интерфейсе. Эта проблема частично решается с помощью настройки рабочего окружения, которая делает его визуально похожим на OllyDbg. К сожалению, большинство возможностей WinDbg всё равно будут доступны только из командной строки.

Для настройки рабочего окружения WinDbg выполните следующие действия:

1. Скачайте архив с настройкой.
2. Распакуйте полученный архив `windbg-workspace-master.zip` в папку `themes` (темы) отладчика. Путь к ней по умолчанию:

```
C:\Program Files (x86)\Windows Kits\8.1\Debuggers\x64\themes.
```

3. Среди скопированных файлов найдите и запустите `windbg.reg`. Затем нажмите кнопку “Yes” в диалоге подтверждения.

После настройки окно WinDbg будет выглядеть как на иллюстрации 3-1.

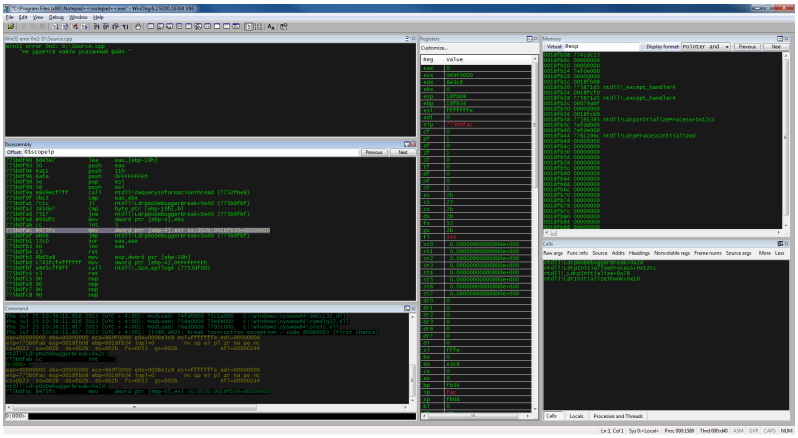


Иллюстрация 3-1. Главное окно WinDbg после настройки рабочего окружения

Инструменты для анализа памяти

Помимо отладчика нам понадобится приложение для анализа памяти запущенного процесса.

Инструмент с открытым исходным кодом **Cheat Engine** предоставляет функции сканера памяти, отладчика и **Hex-редактора** (редактор бинарных файлов). Мы будем использовать Cheat Engine в основном как сканер для поиска адреса переменной в памяти процесса и модификации её значения. Более подробно этот инструмент описан в руководстве пользователя.

HeapMemView – бесплатный инструмент для анализа сегментов динамической памяти (heap, иногда переводится как “куча”), выделенных процессом. HeapMemView имеет две версии: для 32 и 64-битных приложений. В некоторых случаях он будет нам полезен.

Организация памяти процесса

Организация памяти процессов ОС Windows рассмотрена во многих книгах и статьях. Мы изучим только те аспекты этого вопроса, которые имеют отношение к поиску переменных в памяти, а также чтению и записи их значений.

Адресное пространство процесса

Исполняемый EXE-файл и запущенный процесс ОС – это не одно и то же. Файл – это некоторые данные, записанные на устройство хранения информации (например жёсткий диск). Исполняемый файл содержит инструкции (или машинный код), которые выполняет процессор без каких либо дополнительных преобразований.

Когда вы запускаете EXE-файл, для его исполнения ОС нужно выполнить несколько шагов. Во-первых, прочитать его содержимое с устройства хранения и записать в **оперативную память** (random-access memory или RAM). Благодаря этому процессор получает намного более быстрый доступ к инструкциям из файла, поскольку скорость его интерфейса с RAM на несколько порядков выше чем с любым диском.

Когда содержимое файла записано в оперативную память, ОС загружает туда же все необходимые для его работы динамические библиотеки. После этого шага, процесс готов к выполнению. Поскольку все современные ОС для компьютеров и телефонов многозадачные, несколько процессов могут исполняться параллельно. Параллельность в данном случае не означает одновременность. То есть если у компьютера один процессор с одним ядром, он будет переключаться между процессами. В таком случае говорят о распределении процессорного времени. В многозадачных ОС этим занимается специальная программа **планировщик** (scheduler). Благодаря ей каждый процесс получает единицы времени (тики или секунды) в зависимости от своего приоритета.

Чем занимается запущенный процесс? Чтобы ответить на этот вопрос, заглянем в типичный исполняемый файл. В основном он содержит алгоритмы обработки и интерпретации каких-то данных. Следовательно, большая часть работы процесса заключается в манипуляции данными.

Где процесс хранит свои данные? Мы уже знаем, что ОС всегда загружает исполняемые инструкции в оперативную память. В случае данных, сам процесс может свободно выбрать место их хранения: жёсткий диск, оперативная память или даже удалённый компьютер (например игровой сервер подключённый по сети). Большая часть данных, необходимых во время работы процесса копируются в оперативную память для ускорения доступа к ней. Поэтому, именно в RAM мы можем прочитать состояния игровых объектов. Они будут доступны на протяжении всего времени выполнения (runtime) процесса.

Иллюстрация 3-2 демонстрирует элементы типичного процесса. Как правило, он состоит из нескольких модулей. Обязательным из них является EXE, который содержит все инструкции и данные, загруженные из исполняемого файла. Другие модули (обозначенные DLL_1 и DLL_2) соответствуют библиотекам, функции которых вызываются из EXE.

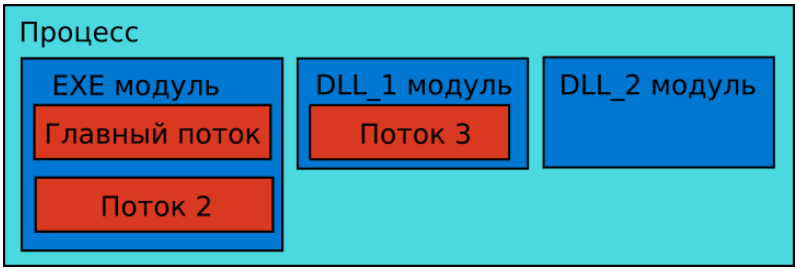


Иллюстрация 3-2. Элементы типичного процесса Windows

Все Windows приложения используют как минимум одну системную библиотеку, которая предоставляет доступ к WinAPI-функциям. Даже если вы не пользуетесь WinAPI явно в своей программе, компилятор вставляет вызовы `ExitProcess` и `VirtualQuery` автоматически в ходе компиляции. Они отвечают за корректное завершение процесса и управление его памятью.

Мы рассмотрели исполняемый файл и запущенный процесс. Теперь поговорим о библиотеках с функциями. Они делятся на два типа: **динамически подключаемые** (dynamic-link libraries или DLL) и **статически подключаемые** (static libraries). Главное различие между ними заключается во времени разрешения зависимостей. Когда исполняемый файл использует функцию библиотеки, говорят, что он от неё зависит.

Статически подключаемые библиотеки должны быть доступны в момент компиляции. Программа **компоновщик** собирает их и исполняемый файл в единый выходной файл. Таким образом, EXE-модуль на иллюстрации 3-2 содержит машинный код и статических библиотек, и исполняемого файла.

Динамически подключаемые библиотеки также должны быть доступны в момент компиляции. Однако, результирующий файл на выходе компоновщика не содержит их машинный код. Вместо этого ОС ищет и загружает эти DLL библиотеки в момент запуска приложения. Если найти их не удалось, приложение завершает свою работу с ошибкой. На иллюстрации 3-2 у процесса есть два модуля DLL, соответствующие динамическим библиотекам.

Рассмотрим, как CPU выполняет инструкции процесса. Эти инструкции – элементарные шаги более сложных высокоуровневых алгоритмов. Результат выполнения каждого шага сохраняется в регистрах (или ячейках памяти) процессора и используется в дальнейшем или выгружается в оперативную память.

Запущенное приложение может использовать несколько алгоритмов в ходе своей работы. Некоторые из них могут выполняться параллельно (так же как процессы в многозадачной ОС). **Поток** (thread) – это часть машинного кода процесса, которая может выполняться независимо от других частей. Потоки взаимодействуют друг с другом (обмениваются информацией) через разделяемые ресурсы, например файл

или область RAM. За выбор потока для исполнения в данный момент отвечает уже знакомый нам планировщик ОС. Как правило, число одновременно работающих потоков определяется числом ядер процессора. Но есть технологии (например hyper-threading от Intel), позволяющие более эффективно использовать мощности процессора и исполнять сразу два потока на одном ядре.

Иллюстрация 3-2 демонстрирует, что модули процесса могут содержать несколько потоков, а могут не содержать ни одного. EXE-модуль всегда имеет главный поток (main thread), который первым получает управление при старте приложения.

Рассмотрим структуру памяти типичного процесса. Иллюстрация 3-3 демонстрирует **адресное пространство** процесса, состоящего из EXE-модуля и DLL-библиотеки. Адресное пространство – это множество всех доступных процессу адресов памяти. Оно разделено на блоки, называемые **сегментами**. У каждого из них есть **базовый адрес**, длина и набор прав доступа (на запись, чтение и исполнение). Разделение на сегменты упрощает задачу контроля доступа к памяти. С их помощью ОС может оперировать блоками памяти, а не отдельными адресами.

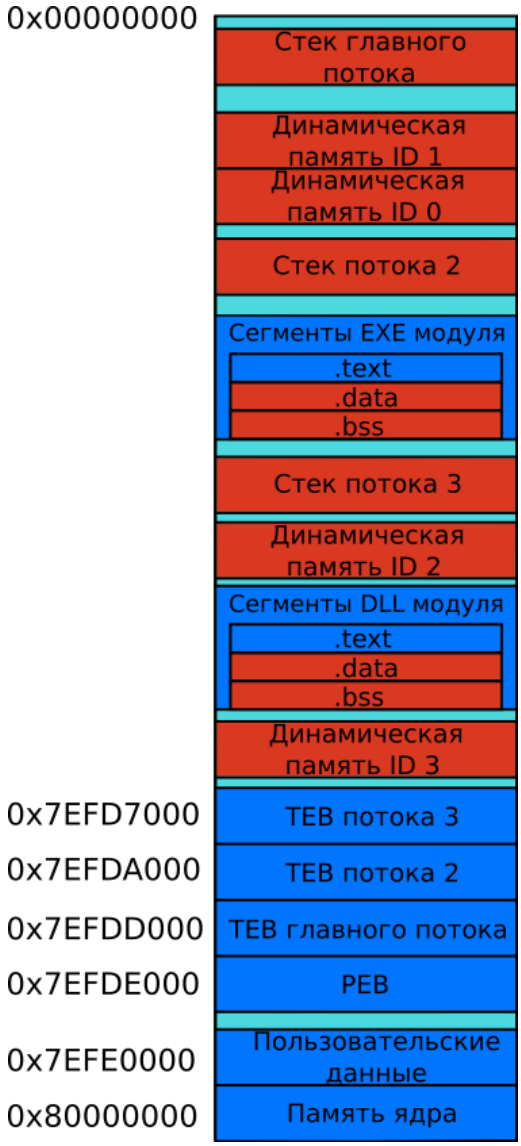


Иллюстрация 3-3. Адресное пространство типичного процесса

Процесс на иллюстрации 3-3 имеет три потока (включая главный). У каждого потока есть свой **сегмент стека**. Стек – это область памяти, организованная по принципу “последним пришёл — первым вышел” (“last in — first out” или LIFO). Она инициализируется ОС при старте приложения и используется для хранения переменных и вызова функций. В стеке сохраняется адрес инструкции, следующей за вызовом. После возврата из функции процесс продолжает своё выполнение с этой инструкции. Также через стек передаются входные параметры функций.

Кроме сегментов стека, у процесса есть несколько **сегментов динамической памяти** (heap), к которым имеет доступ каждый поток.

У всех модулей процесса есть обязательные сегменты: `.text`, `.data` и `.bss`. Кроме обязательных могут быть и дополнительные сегменты (например `.rsrc`). Они не представлены на схеме 3-3.

Таблица 3-1 кратко описывает каждый сегмент из иллюстрации 3-3. Во втором столбце приведены их обозначения в отладчике OllyDbg.

Таблица 3-1. Описание сегментов

Сегмент	Обозначение в OllyDbg	Описание
Стек главного потока	Stack of main thread	Содержит автоматические переменные (память под которые выделяется при входе в блок области видимости и освобождается при выходе из него), стек вызовов с адресами возврата из функций и их входные параметры.
Динамическая память ID 1	Heap	Дополнительный сегмент памяти, который создаётся при переполнении сегмента динамической памяти ID 0.
Динамическая память ID 0	Default heap	ОС всегда создаёт этот сегмент при запуске процесса. Он используется по умолчанию для хранения переменных.
Стек потока 2	Stack of thread 2	Выполняет те же функции, что и стек главного потока, но используется только потоком 2.
`.text` EXE модуля	Code	Содержит машинный код модуля EXE.
`.data` EXE модуля	Data	Содержит статические и не константные глобальные переменные модуля EXE, которые инициализируются значениями при создании.
`.bss` EXE модуля		Содержит статические и не константные глобальные переменные модуля EXE, которые не инициализируются при создании.
Стек потока 3	Stack of thread 2	То же самое, что и стек потока 2, только используется потоком 3.
Динамическая память ID 2		Дополнительный сегмент памяти, расширяющий сегмент динамической памяти ID 1 при его переполнении.
`.text` модуля DLL	Code	Содержит машинный код модуля DLL.
`.data` модуля DLL	Data	Содержит статические и не константные глобальные переменные модуля DLL, которые инициализируются значениями при создании.
`.bss` модуля DLL		Содержит статические и не константные глобальные переменные модуля DLL, которые не инициализируются при создании.
Динамическая память ID 3		Дополнительный сегмент памяти, расширяющий сегмент динамической памяти ID 2 при его переполнении.

Сегмент	Обозначение в OllyDbg	Описание
TEB потока 3	Data block of thread 3	Содержит **блок информации о потоке** (Thread Information Block или TIB), также известный как **блок контекста потока** (Thread Environment Block или TEB). Он представляет собой структуру с информацией о потоке 3.
TEB потока 2	Data block of thread 2	Содержит TEB структуру потока 2.
TEB главного потока	Data block of main thread	Содержит TEB структуру главного потока.
PEB	Process Environment Block	Содержит **блок контекста процесса** (Process Environment Block или PEB). Эта структура данных с информацией о процессе в целом.
Пользовательские данные	User Share Data	Содержит данные, которые доступны и совместно используются текущим процессом и другим.
Память ядра	Kernel memory	Область памяти, зарезервированная для нужд ОС.

Предположим, что на иллюстрации 3-3 приведено адресное пространство процесса игрового приложения. В этом случае состояние игровых объектов может находится в сегментах, отмеченных красным цветом.

ОС назначает базовые адреса этих сегментов в момент старта приложения. Эти адреса могут отличаться от запуска к запуску. Кроме того, последовательность сегментов в памяти может также меняться. В то же время некоторые из сегментов, отмеченных синим цветом на иллюстрации 3-3 (например PEB, User Share Data и Kernel memory), имеют неизменный адрес при каждом старте приложения.

Отладчик OllyDbg позволяет прочитать структуру памяти (memory map) запущенного процесса. Иллюстрации 3-4 и 3-5 демонстрируют вывод OllyDbg для приложения, адресное пространство которого приведено на схеме 3-3.

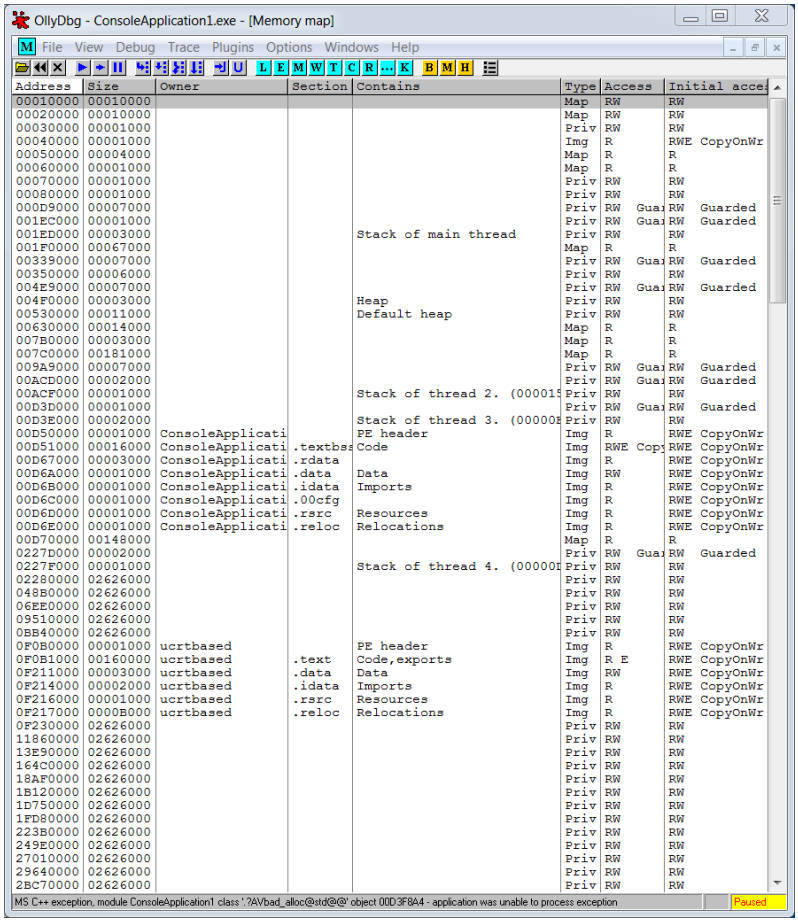


Иллюстрация 3-4. Структура памяти процесса в OllyDbg

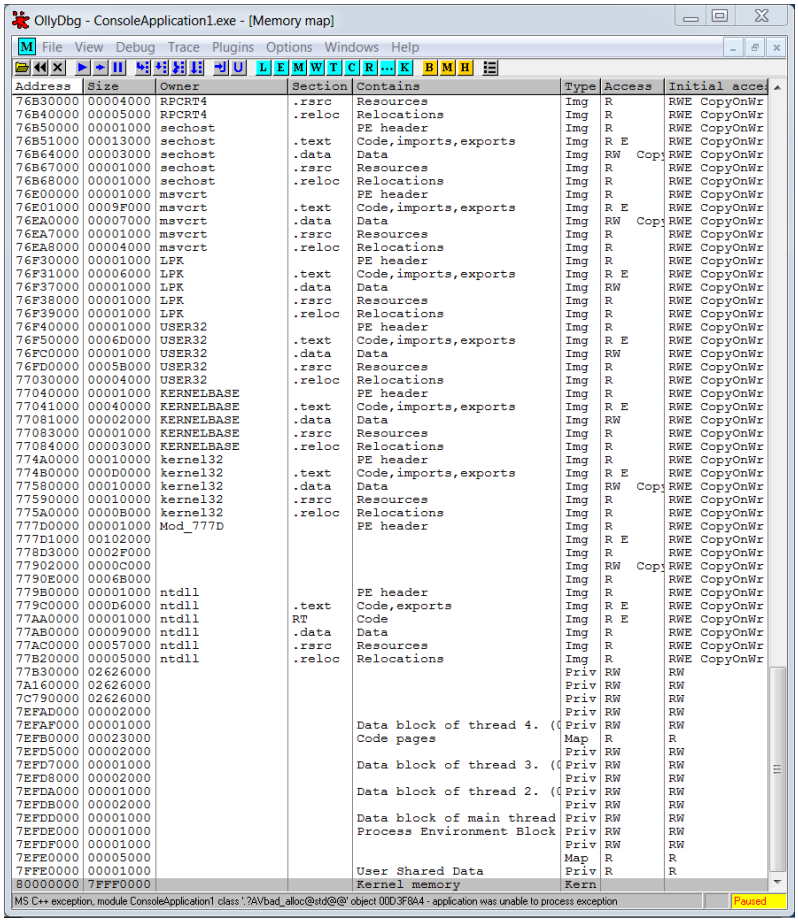


Иллюстрация 3-5. Структура памяти процесса в OllyDbg (продолжение)

Таблица 3-2 демонстрирует соответствие между схемой 3-3 и сегментами настоящего процесса из иллюстраций 3-4 и 3-5.

Таблица 3-2. Сегменты процесса

Базовый адрес	Сегмент	Обозначение в OllyDbg
001ED000	Стек главного потока	Stack of main thread
004F0000	Динамическая память ID 1	Heap
00530000	Динамическая память ID 0	Default heap
00ACF000	Стеки вспомогательных	Stack of thread N
00D3E000	потоков	
0227F000		
00D50000-00D6E000	Сегменты EXE модуля “ConsoleApplication1”	
02280000-0BB40000	Дополнительные сегменты	
0F230000-2BC70000	динамической памяти	
0F0B0000-0F217000	Сегменты модуля DLL “ucrtbased”	
7EFAF000	TEB вспомогательных	Data block of thread N
7EFD7000	потоков	
7EFDA000		
7EFDD000	TEB главного потока	Data block of main thread
7EFDE000	PEB главного потока	Process Environment Block
7FFE0000	Пользовательские данные	User shared data
80000000	Память ядра	Kernel memory

Возможно, вы обратили внимание, что OllyDbg не может автоматически идентифицировать все сегменты динамической памяти. С этой задачей лучше справляются отладчик WinDbg и инструмент HeapMemView.

Поиск переменной в памяти

Внутриигровые боты читают состояния объектов из памяти процесса игрового приложения. Эти состояния могут храниться в нескольких переменных, находящихся в разных сегментах. Базовые адреса этих сегментов и смещение переменных внутри них могут меняться от запуска к запуску. Это означает, что абсолютные адреса переменных непостоянны. К сожалению, бот может читать данные из памяти только по абсолютным адресам. Следовательно, он должен уметь искать нужные ему переменные самостоятельно.

Термин “абсолютный адрес” неточен, если мы говорим о **модели сегментации памяти x86**. x86 – это архитектура процессора, впервые реализованная компанией Intel. Сегодня практически все настольные компьютеры имеют процессоры этой архитектуры. Правильный термин, который следует употреблять – “линейный адрес”. Он вычисляется по следующей формуле:

```
1 линейный адрес = базовый адрес сегмента + смещение в сегменте
```

В этой главе мы продолжим использовать термин “абсолютный адрес”, поскольку он интуитивно понятен.

Задачу поиска переменной в памяти процесса можно разделить на три этапа. В результате получится следующий алгоритм:

1. Найти сегмент, который содержит искомую переменную.
2. Определить базовый адрес сегмента.
3. Определить смещение переменной внутри сегмента.

Очень высока вероятность того, что переменная будет храниться в одном и том же сегменте при каждом старте приложения. Это правило не выполняется для сегментов динамической памяти, что связано с особенностью её организации. Если мы установили, что переменная не находится в сегменте динамической памяти, первый шаг алгоритма может быть выполнен вручную. Полученный результат можно закодировать в боте без каких-либо дополнительных условий и проверок. В противном случае бот должен искать сегмент самостоятельно.

Второй шаг алгоритма бот должен всегда выполнять сам. Как мы упоминали ранее, адреса сегментов меняются при старте приложения.

Последний шаг алгоритма – найти смещение переменной в сегменте. Нет никаких гарантий, что оно не будет меняться при каждом старте приложения. Однако, смещение может оставаться тем же в некоторых случаях. Это зависит от типа сегмента, как демонстрирует таблица 3-3. Таким образом, в некоторых случаях мы можем выполнить третий шаг алгоритма вручную и закодировать результат в боте.

Таблица 3-3. Смещение переменных в различных типах сегментов

Тип сегмента	Постоянство смещения
`.bss`	Смещение переменной не меняется
`.data`	при перезапуске приложения.
Стек	В большинстве случаев смещение переменной не меняется. Но оно зависит от **порядка выполнения инструкций** (control flow). Если этот порядок меняется, смещение, скорее всего, тоже изменится.
Динамическая память	Смещение переменной меняется при перезапуске приложения.

Поиск переменной в 32-битном приложении

Применим алгоритм поиска переменной на практике. Выполним все его шаги вручную для приложения ColorPix, которым мы пользовались в прошлой главе для чтения цветов и координат пикселей экрана. Это поможет лучше понять и запомнить все необходимые действия.

Приложение ColorPix является 32-битным. Скриншот его окна приведён на иллюстрации 3-6. Попробуем найти в памяти переменную, которая соответствует координате X выделенного на экране пикселя. На иллюстрации 3-6 она подчёркнута красной линией.

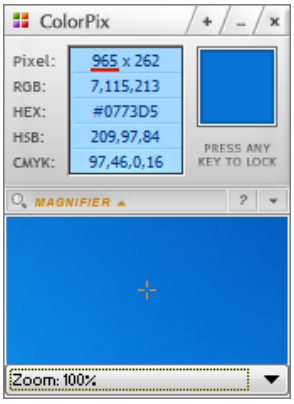


Иллюстрация 3-6. Окно приложения ColorPix

В ходе дальнейших действий вы не должны закрывать уже запущенное приложение ColorPix. Иначе, вам придётся начать поиск переменной сначала.

Для начала найдём сегмент памяти, в котором хранится переменная. Эту задачу можно разделить на два этапа:

1. Найти абсолютный адрес переменной с помощью сканера памяти Cheat Engine.
2. Сравнить найденный адрес с базовыми адресами всех сегментов. Таким образом мы узнаем сегмент, в котором хранится переменная.

Чтобы найти переменную с помощью Cheat Engine, выполните следующие действия:

1. Запустите 32-битную версию сканера с правами администратора.
2. Выберите пункт главного меню “File” -> “Open Process”. Вы увидите диалог со списком запущенных процессов (см. иллюстрацию 3-7).

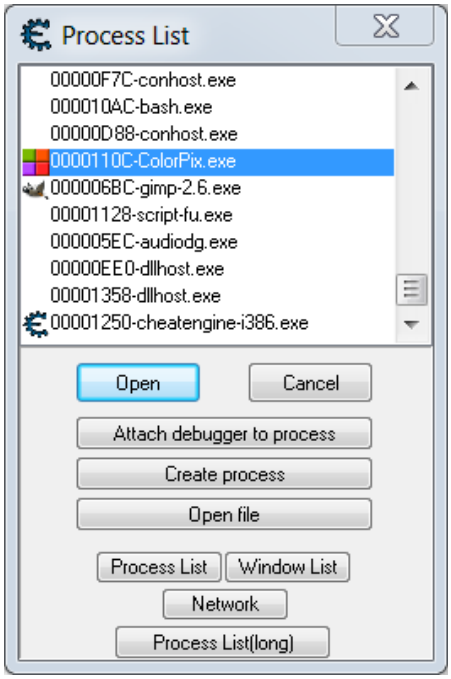


Иллюстрация 3-7. Диалог выбора процесса Cheat Engine

3. Выберите процесс с именем “ColorPixel.exe” и нажмите кнопку “Open”. В результате имя этого процесса отобразится в верхней части окна Cheat Engine.
4. Введите значение координаты X, которое вы видите в данный момент в окне ColorPixel, в поле “Value” окна Cheat Engine.
5. Нажмите кнопку “First Scan”, чтобы найти абсолютный адрес указанного значения координаты X в памяти процесса ColorPixel.

Когда вы нажимаете кнопку “First Scan”, значение в поле “Value” окна Cheat Engine, должно соответствовать тому, что отображает ColorPixel. Координата X изменится, если вы переместите курсор мыши по экрану, поэтому нажать на кнопку будет затруднительно. Воспользуйтесь комбинацией клавиш Shift+Tab, чтобы переключиться на неё и Enter, чтобы нажать.

В левой части окна Cheat Engine вы увидите результаты поиска, как на иллюстрации 3-8.

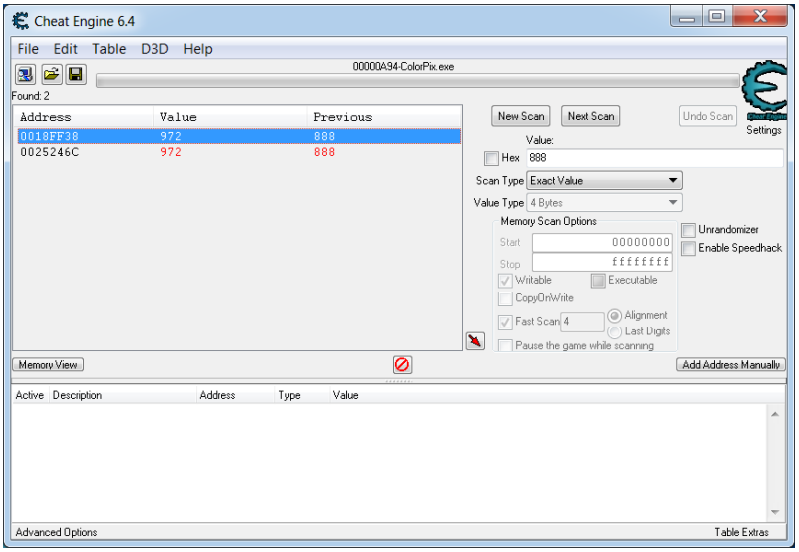


Иллюстрация 3-8. Результаты поиска в окне Cheat Engine

Если в момент сканирования процесса несколько переменных имеют то же самое значение что и координата X, найденных переменных будет больше чем две. В этом случае вам надо отфильтровать ошибочные результаты. Для этого выполните следующие шаги:

1. Переместите курсор мыши, чтобы значение координаты X в окне ColorPixel изменилось.
2. Введите новую координату X в поле “Value” окна Cheat Engine.
3. Нажмите кнопку “Next Scan”.

После этого в окне результатов должны остаться только две переменные, как на иллюстрации 3-8. В моём случае их абсолютные адреса равны 0018FF38 и 0025246C. У вас они могут отличаться, но это не существенно для нашего примера.

Мы нашли абсолютные адреса двух переменных, хранящих значение координаты X. Теперь определим сегменты, в которых они находятся. Для этой цели воспользуемся отладчиком OllyDbg. Для поиска сегментов выполните следующие шаги:

1. Запустите отладчик OllyDbg с правами администратора. Путь к нему по умолчанию:
`C:\Program Files (x86)\odbg201\ollydbg.exe`.

2. Выберите пункт главного меню “File” -> “Attach”. Вы увидите диалог со списком запущенных 32-битных процессов (см. иллюстрацию 3-9).

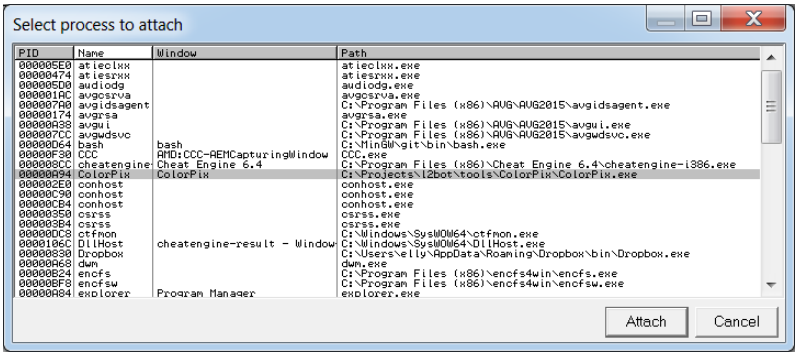


Иллюстрация 3-9. Диалог выбора процесса в отладчике OllyDbg

3. Выберите процесс “ColorPix” в списке и нажмите кнопку “Attach”. Когда отладчик подключится к нему, вы увидите состояние “Paused” в правом нижнем углу окна OllyDbg.
4. Нажмите комбинацию клавиш Alt+M, чтобы открыть окно, отображающее структуру памяти процесса ColorPix. Это окно “Memory Map” приведено на иллюстрации 3-10.

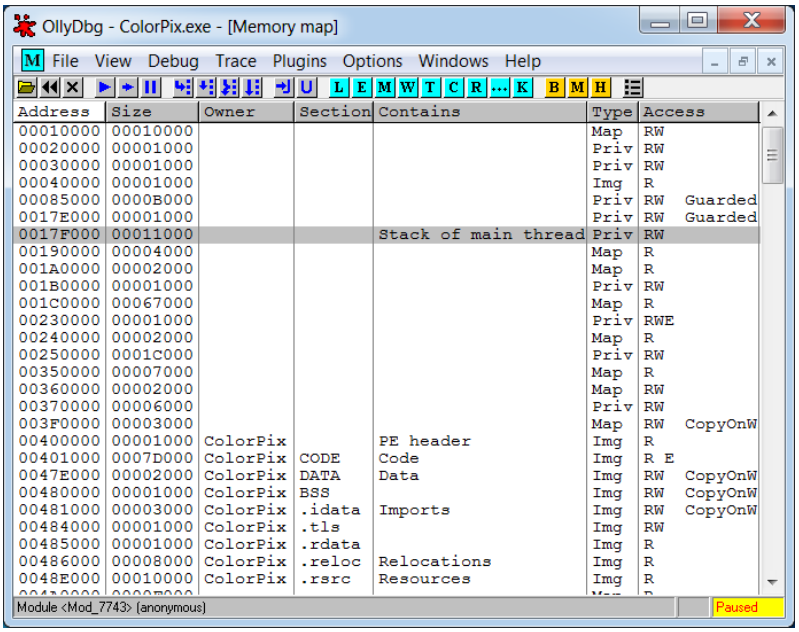


Иллюстрация 3-10. Окно Memory Map со структурой памяти процесса

Переменная с абсолютным адресом 0018FF38 хранится в сегменте стека главного процесса (“Stack of main thread”), который занимает адреса с 0017F000 по 00190000.

OllyDbg отображает только адрес начала сегмента и его размер. Чтобы вычислить конечный адрес, вы должны сложить два эти числа. Результат будет равен адресу начала следующего сегмента.

Вторая найденная нами переменная с адресом 0025246C находится в сегменте с базовым адресом 00250000, тип которого неизвестен. Найти его будет труднее чем сегмент стека. Поэтому мы продолжим работу с первой переменной.

Последний шаг поиска – расчёт смещения переменной в сегменте стека. Стек в архитектуре x86 растёт вниз. Это означает, что он начинается с больших адресов и расширяется в сторону меньших. Следовательно, базовый адрес стека равен его верхней границе (в нашем случае это 00190000). Нижняя границе стека может меняться по ходу его увеличения.

Смещение переменной равно разности базового адреса сегмента, в котором она находится, и её абсолютного адреса. В нашем случае мы получим:

```
1 00190000 - 0018FF38 = C8
```

Для сегментов динамической памяти, `.bss` и `.data` это вычисление выглядело бы иначе. Все они растут вверх (в сторону больших адресов), поэтому их базовый адрес соответствует нижней границе.

Теперь у нас есть вся необходимая информация, чтобы найти и прочитать координату X в любом запущенном процессе ColorPix. Алгоритм бота, который бы это делал, выглядит следующим образом:

1. Прочитать базовый адрес сегмента стека главного потока. Этот адрес хранится в TEB сегменте.
2. Вычесть смещение переменной (всегда равное C8) из базового адреса сегмента стека. В результате получим её абсолютный адрес.
3. Прочитать значение переменной из памяти процесса ColorPix по её абсолютному адресу.

Корректность первого шага алгоритма мы можем проверить вручную с помощью отладчика OllyDbg. Он позволяет прочитать информацию сегмента TEB в удобном виде. Для этого дважды щёлкните по сегменту, который называется “Data block of main thread”, в окне “Memory Map” отладчика. Вы увидите окно как на иллюстрации 3-11.

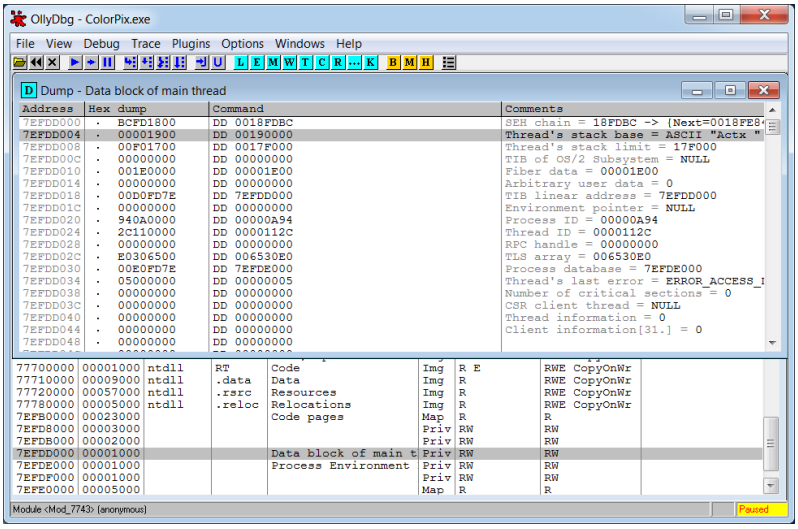


Иллюстрация 3-11. Окно OllyDbg с информацией TEB

Базовый адрес сегмента стека 00190000 указан во второй строчке открывшегося окна. Учтите, что этот адрес может меняться при каждом запуске приложения.

Поиск переменной в 64-битном приложении

Применим наш алгоритм поиска переменной для 64-битного приложения.

Отладчик OllyDbg не поддерживает 64-битные приложения, поэтому вместо него воспользуемся WinDbg.

Resource Monitor (монитор ресурсов) Windows 7 будет нашим приложением для анализа. Он распространяется вместе с ОС и доступен сразу после её установки. Разрядность Resource Monitor совпадает с разрядностью Windows. Чтобы запустить приложение, откройте меню Пуск (Start) Windows и введите следующую команду в строку поиска:

```
1 perfmon.exe /res
```

Иллюстрации 3-12 демонстрирует окно Resource Monitor.

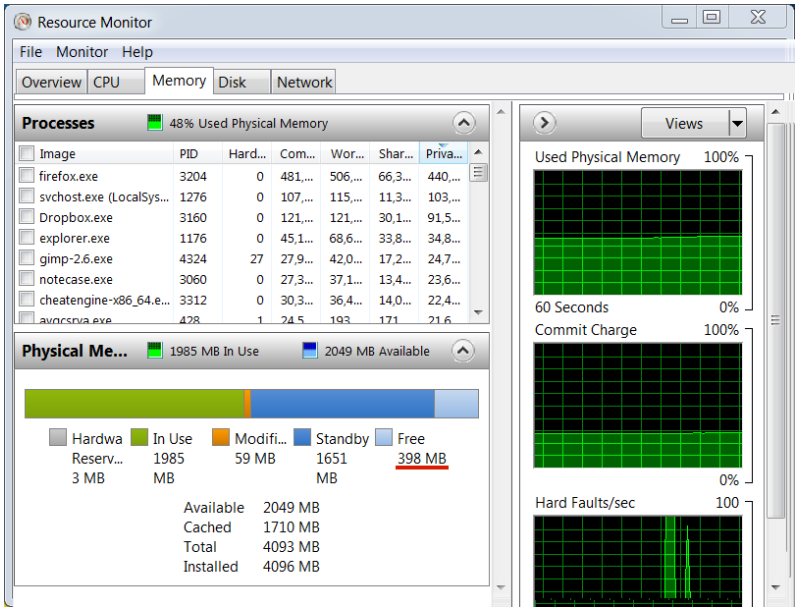


Иллюстрация 3-12. Окно приложения Resource Monitor

Найдём переменную, хранящую размер свободной памяти системы. На иллюстрации её значение подчёркнуто красной линией.

Прежде всего найдём сегмент, содержащий искомую переменную. Для этого воспользуемся 64-битной версией сканера Cheat Engine. Интерфейс его 64 и 34-битных версий одинаков, поэтому вам нужно выполнить те же действия, что и при анализе приложения ColorPixel.

В моем случае сканер нашёл две переменные с адресами 00432FEC и 00433010. Определим сегменты, в которых они хранятся. Чтобы прочитать структуру памяти процесса с помощью отладчика WinDbg, выполните следующие действия:

1. Запустите 64-битную версию WinDbg с правами администратора. Путь к нему по умолчанию:
`C:\Program Files (x86)\Windows Kits\8.1\Debuggers\x64\windbg.exe`.

2. Выберите пункт главного меню “File” -> “Attach to a Process…”. Откроется окно диалога со списком запущенных 64-разрядных процессов, как на иллюстрации 3-13.

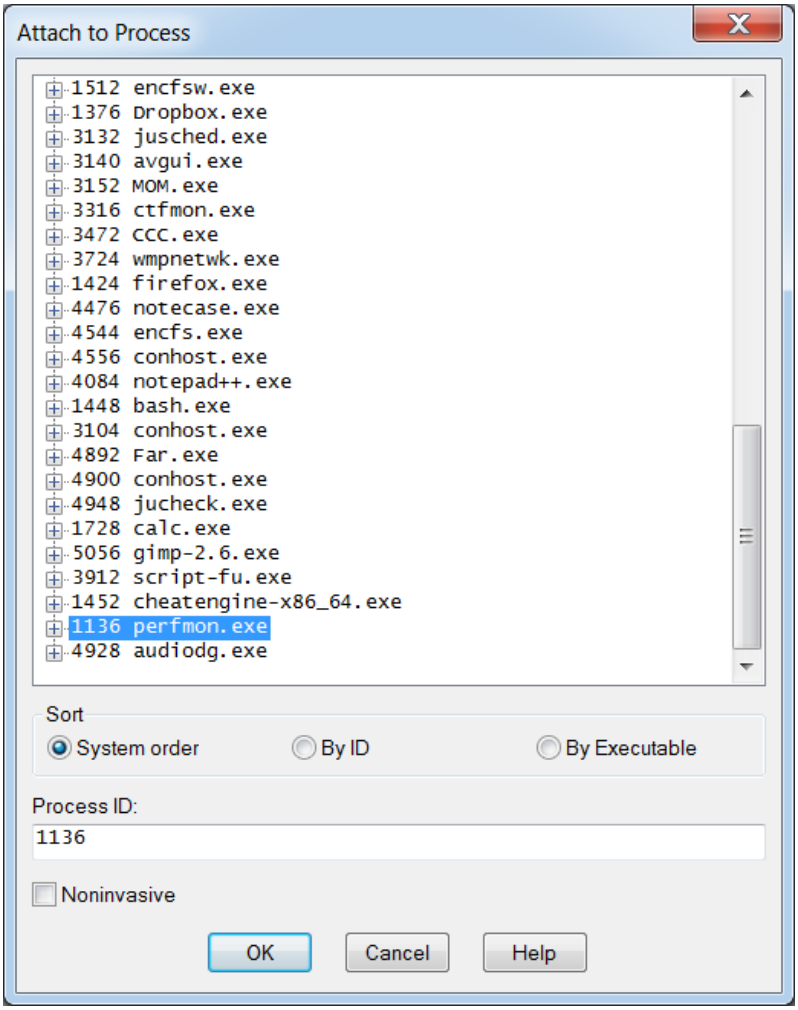


Иллюстрация 3-13. Диалог выбора процесса в отладчике WinDbg

3. Выберите в списке процесс “perfmon.exe” и нажмите кнопку “OK”.
4. В командной строке отладчика, расположенной в нижней части окна “Command”, введите текст `!address` и нажмите Enter. Структура памяти процесса отобразится в окне “Command”, как на иллюстрации 3-14.

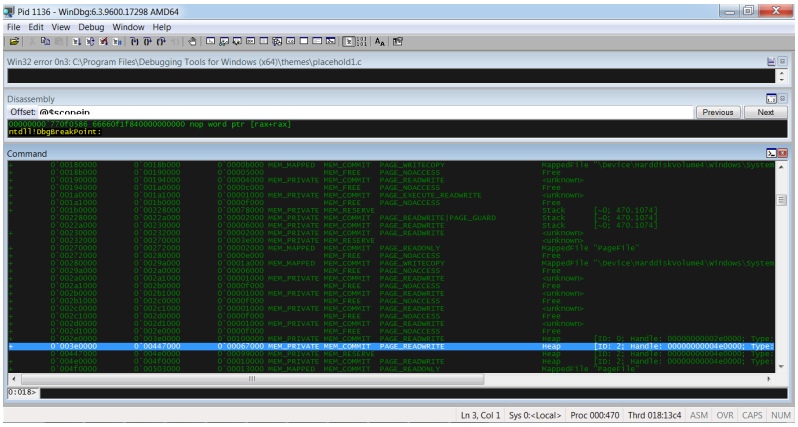


Иллюстрация 3-14. Вывод структуры памяти процесса в окне Command

Обе переменные с абсолютными адресами 00432FEC и 00433010 находятся в сегменте динамической памяти с ID 2. Границы этого сегмента: с 003E0000 по 00447000. Смещение первой переменной в сегменте равно 52FEC:

```
1 00432FEC - 003E0000 = 52FEC
```

Задача решена.

Для бота алгоритм поиска переменной, хранящей размер свободной памяти ОС в приложении Resource Monitor, выглядит следующим образом:

1. Прочитать базовый адрес сегмента динамической памяти с ID 2. Чтобы получить доступ к этим сегментам, надо воспользоваться следующими WinAPI-функциями:
   * CreateToolhelp32Snapshot
   * Heap32ListFirst
   * Heap32ListNext
2. Добавить смещение переменной (в моем случае равное 52FEC) к базовому адресу сегмента. В результате получится её абсолютный адрес.
3. Прочитать значение переменной из памяти процесса.

Как вы помните, смещение переменной в сегменте динамической памяти обычно меняется при перезапуске приложения. В случае если приложение достаточно простое (как рассматриваемый нами Resource Monitor), порядок выделения динамической памяти может быть одним и тем же при каждом старте программы.

Попробуйте перезапустить Resource Monitor и найти переменную ещё раз. Вы получите то же самое её смещение в сегменте, равное 52FEC.

Выводы

Мы рассмотрели адресное пространство Windows процесса. Затем составили алгоритм поиска переменной в памяти и применили его к 32 и 64-разрядному приложениям. В ходе этого мы познакомились с функциями отладчиков OllyDbg и WinDbg для анализа структуры памяти процесса.

Доступ к памяти процесса

Мы научились вручную искать переменные в памяти процесса. Пришло время написать код, автоматизирующий эту задачу. К сожалению, внутриигровые боты не могут использовать программу-отладчик (например OllyDbg). Вместо этого все необходимые возможности должны быть реализованы в их коде.

Подключение к процессу

Как вы помните, перед началом работы с памятью процесса к нему нужно подключить отладчик. После этого он получает полный доступ к адресному пространству процесса. Мы выполняли это действие через диалог интерфейса пользователя. То же самое должен уметь внутриигровой бот. Рассмотрим, какими WinAPI-функциями он может для этого воспользоваться.

Практически все объекты и ресурсы Windows доступны через их дескрипторы. WinAPI-функция `OpenProcess` позволяет получить дескриптор работающего процесса. Возникает вопрос: как сообщить ОС, что нас интересует именно процесс игрового приложения? Для этой цели служит **идентификатор процесса** (process identifier или PID). PID – это уникальный номер, который ОС присваивает каждому процессу при старте. Его мы должны передать в `OpenProcess` входным параметром. Далее получив дескриптор процесса, мы можем обращаться к его памяти с помощью других WinAPI-функций.

Windows отвечает за распределение своих ресурсов между запущенными процессами. Один из этих ресурсов – память. Если любой процесс всегда будет иметь доступ к памяти других процессов, это может привести к сбоям в их работе. Система в целом будет ненадёжна. Поэтому ОС имеет специальный механизм защиты доступа к своим объектам. Рассмотрим его подробнее.

В архитектуре Windows разработчику пользовательских приложений предоставляются высокоуровневые абстракции к ресурсам ОС. Объекты Windows (например процессы) также используют эти абстракции. Другими словами, одни объекты служат обёртками для системных ресурсов и предоставляют к ним единообразный интерфейс для других объектов. Такой подход упрощает интерфейсы для разработки как системных библиотек Windows, так и пользовательских приложений.

Представим, что мы разрабатываем пользовательское приложение, например внутриигрового бота. Каким образом оно может взаимодействовать с каким-нибудь

Windows объектом? Каждый объект представляет собой структуру, состоящую из **заголовка** (header) и **тела** (body). Тело содержит данные, специфичные для каждого типа объектов. Заголовок же включает метаинформацию, которая используется **менеджером объектов** (Object Manager). Именно он предоставляет доступ к ресурсам ОС через соответствующие им объекты.

Модель безопасности Windows ограничивает процессам доступ к системным объектам и различным действиям, требующих прав администратора. Можно сказать, что менеджер объектов реализует модель безопасности Windows. Согласно ей, процесс должен иметь специальные привилегии, чтобы получить доступ к памяти другого через вызов `OpenProcess`. Управлять привилегиями процесса можно с помощью специального объекта Windows под названием **маркер доступа** (access token).

Учитывая модель безопасности Windows, полный алгоритм подключения к процессу через WinAPI-функцию `OpenProcess` выглядит следующим образом:

1. Получить дескриптор текущего процесса.
2. По дескриптору получить маркер доступа текущего процесса.
3. Предоставить привилегию `SE_DEBUG_NAME` для маркера доступа. Эта привилегия даёт право отлаживать другие процессы.
4. Получить дескриптор целевого процесса через вызов `OpenProcess`.

Приложение, реализующее этот алгоритм, должно быть запущено с правами администратора. Без них невозможно выполнить третий шаг и предоставить текущему процессу привилегию `SE_DEBUG_NAME` через WinAPI-функцию `AdjustTokenPrivileges`.

Вам может показаться странным, что приложению, запущенному с правами администратора, надо предоставлять дополнительные права на отладку других процессов. В самом деле, логично предположить, что администратору системы по умолчанию должны быть доступны все её возможности. Но это не означает, что любое запущенное им приложение должно нарушать модель безопасности Windows. Такое поведение может привести к нестабильной работе всей системы.

Листинг 3-1 демонстрирует код приложения, которое подключается к процессу с заданным PID.

Листинг 3-1. Приложение `OpenProcess.cpp`

```
#include <windows.h>
#include <stdio.h>

BOOL SetPrivilege(HANDLE hToken, LPCTSTR lpszPrivilege, BOOL bEnablePrivilege)
{
    TOKEN_PRIVILEGES tp;
    LUID luid;
    if (!LookupPrivilegeValue(NULL, lpszPrivilege, &luid))

    {
        printf("LookupPrivilegeValue error: %u\n", GetLastError());
```

```
        return FALSE;
    }

    tp.PrivilegeCount = 1;
    tp.Privileges[0].Luid = luid;

    if (bEnablePrivilege)
        tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
    else
        tp.Privileges[0].Attributes = 0;

    if (!AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(TOKEN_PRIVILEGES),
                               (PTOKEN_PRIVILEGES)NULL, (PDWORD)NULL))
    {
        printf("AdjustTokenPrivileges error: %u\n", GetLastError());
        return FALSE;
    }

    if (GetLastError() == ERROR_NOT_ALL_ASSIGNED)
    {
        printf("The token does not have the specified privilege. \n");
        return FALSE;
    }
    return TRUE;
}

int main()
{
    HANDLE hProc = GetCurrentProcess();
    HANDLE hToken = NULL;

    if (!OpenProcessToken(hProc, TOKEN_ADJUST_PRIVILEGES, &hToken))
        printf("Failed to open access token\n");

    if (!SetPrivilege(hToken, SE_DEBUG_NAME, TRUE))
        printf("Failed to set debug privilege\n");

    DWORD pid = 1804;

    HANDLE hTargetProc = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
    if (hTargetProc)
        printf("Target process handle = %p\n", hTargetProc);
    else
        printf("Failed to open process: %u\n", GetLastError());

    CloseHandle(hTargetProc);
    return 0;
}
```

Приложение из листинга 3-1 подключается к процессу с PID равным 1804. Вам нужно заменить его на PID работающего в данный момент процесса. Узнать идентификаторы всех запущенных процессов можно с помощью приложения Task Manager (диспетчер задач). Укажите PID целевого процесса в следующей строке файла `OpenProcess.cpp`:

```
    DWORD pid = 1804;
```

Каждый шаг алгоритма подключения к процессу выполняется отдельной функцией. Все они вызываются из функции `main`, которая получает управление сразу при старте приложения. Рассмотрим её код подробнее.

Сначала мы с помощью WinAPI-функции `GetCurrentProcess` получаем дескриптор текущего процесса и сохраняем его в переменной `hProc`.

Далее вызывается WinAPI-функция `OpenProcessToken`, которая возвращает маркер доступа. В неё мы передаём дескриптор `hProc` и маску доступа `TOKEN_ADJUST_PRIVILEGES`. Благодаря этой маске мы получаем право менять возвращаемый функцией маркер доступа. Его мы сохраняем в переменной `hToken`.

Весь код, предоставляющий привилегию `SE_DEBUG_NAME` маркеру доступа `hToken`, мы реализовали в отдельной функции `SetPrivilege`. Она выполняет два действия:

1. Читает **локальный уникальный идентификатор** (locally unique identifier или LUID) константы, соответствующей привилегии `SE_DEBUG_NAME` с помощью WinApI функции `LookupPrivilegeValue`.
2. Предоставляет маркеру доступа, переданному входным параметром, привилегию `SE_DEBUG_NAME` (указанную по LUID) через WinAPI-функцию `AdjustTokenPrivileges`.

Функция `SetPrivilege` более детально разбирается в статье.

Последнее действие в функции `main` – подключение к целевому процессу, дескриптор которого сохраняется в переменной `hTargetProc`. Для этого мы используем WinAPI-функцию `OpenProcess`. В неё передаются права доступа `PROCESS_ALL_ACCESS` и PID процесса для подключения. После этого вся его память становится доступна по дескриптору `hTargetProc`.

Операции чтения и записи

Мы знаем как получить дескриптор целевого процесса. Теперь рассмотрим способы обращения к его памяти.

WinAPI-функция `ReadProcessMemory` читает данные из указанной области памяти целевого процесса и сохраняет их в память вызывающего процесса. Аналогичная ей функция `WriteProcessMemory` записывает указанные данные в память целевого процесса. Рассмотрим пример использования этих функций.

Тестовое приложение, приведённое в листинге 3-2, записывает шестнадцатеричное значение DEADBEEF по некоторому абсолютному адресу памяти целевого процесса. Затем по этому же адресу происходит чтение. Если запись была успешной, мы прочитаем то же самое значение DEADBEEF.

Листинг 3-2. Приложение `ReadWriteProcessMemory.cpp`

```
#include <stdio.h>
#include <windows.h>

BOOL SetPrivilege(HANDLE hToken, LPCTSTR lpszPrivilege, BOOL bEnablePrivilege)
{
    // Смотрите реализацию этой функции в листинге 3-1
}

DWORD ReadDword(HANDLE hProc, DWORD_PTR address)
{
    DWORD result = 0;
```

```
12
13      if (ReadProcessMemory(hProc, (void*)address, &result, sizeof(result), NULL) == 0)
14      {
15          printf("Failed to read memory: %u\n", GetLastError());
16      }
17      return result;
18  }
19
20  void WriteDword(HANDLE hProc, DWORD_PTR address, DWORD value)
21  {
22      if (WriteProcessMemory(hProc, (void*)address, &value, sizeof(value), NULL) == 0)
23      {
24          printf("Failed to write memory: %u\n", GetLastError());
25      }
26  }
27
28  int main()
29  {
30      HANDLE hProc = GetCurrentProcess();
31
32      HANDLE hToken = NULL;
33      if (!OpenProcessToken(hProc, TOKEN_ADJUST_PRIVILEGES, &hToken))
34          printf("Failed to open access token\n");
35
36      if (!SetPrivilege(hToken, SE_DEBUG_NAME, TRUE))
37          printf("Failed to set debug privilege\n");
38
39      DWORD pid = 5356;
40      HANDLE hTargetProc = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
41      if (!hTargetProc)
42          printf("Failed to open process: %u\n", GetLastError());
43
44      DWORD_PTR address = 0x001E0000;
45      WriteDword(hTargetProc, address, 0xDEADBEEF);
46      printf("Result of reading dword at 0x%llx address = 0x%x\n", address,
47             ReadDword(hTargetProc, address));
48
49      CloseHandle(hTargetProc);
50      return 0;
51  }
```

Абсолютный адрес 001E0000 для записи значения DEADBEEF выбран произвольно. Эту область памяти занимает какой-то сегмент. Операция записи данных в него может привести к аварийному завершению целевого процесса. Поэтому в качестве него
не используйте важные системные службы Windows. Лучше всего для нашего теста подойдёт приложение Notepad.

Для запуска приложения `ReadWriteProcessMemory.cpp` выполните следующие действия:

1. Запустите Notepad.
2. С помощью Task Manager прочитайте PID процесса Notepad.
3. Присвойте этот PID соответствующей переменной в исходном коде приложения `ReadWriteProcessMemory.cpp`:

```
1 DWORD pid = 5356;
```

4. С помощью отладчика WinDbg прочитайте базовый адрес любого сегмента динамической памяти процесса Notepad. Для этого воспользуйтесь уже

знакомой нам командой `!address`.
5. Отключите WinDbg от процесса Notepad с помощью команды `.detach`.
6. Присвойте базовый адрес сегмента динамической памяти переменной `address` в функции `main`:

```
1 DWORD_PTR address = 0x001E0000;
```

Это нужно потому, что писать случайное значение в динамическую память безопаснее, чем в другие сегменты.

7. Скомпилируйте приложение `ReadWriteProcessMemory.cpp`. Разрядность (x86 или x64) полученного EXE-файл должна соответствовать разрядности Notepad. В противном случае наше приложение не сможет к нему подключиться.
8. Запустите тестовое приложение с правами администратора из командной строки Windows.

После успешного выполнения нашего примера, вы увидите в консоли строку:

```
1 Result of reading dword at 0x1e0000 address = 0xdeadbeef
```

В этом выводе указан абсолютный адрес для записи и прочитанное по нему же значение.

Обратите внимание на функции-обёртки `WriteDword` и `ReadDword` в листинге 3-2. Они скрывают несущественные детали и предоставляют простой интерфейс к WinAPI-функциям `WriteProcessMemory` и `ReadProcessMemory`. Их параметры представлены в таблице 3-4.

Таблица 3-4. Параметры функций `WriteProcessMemory` и `ReadProcessMemory`

Номер параметра	Параметр	Описание
1	`hProc`	Дескриптор целевого процесса, к памяти которого идёт обращение.
2	`address`	Абсолютный адрес области памяти для доступа.
3	`result`	Указатель на область памяти текущего процесса, в которую будет сохранён результат вызова `ReadProcessMemory`.
3	`value`	Указатель на буфер данных, которые будут записаны функцией `WriteProcessMemory` в память целевого процесса.
4	`sizeof(...)`	Число байт для чтения или записи.
5	`NULL`	Указатель на переменную. Если операция чтения или записи была прервана по какой-то причине, в эту переменную запишется число переданных байт.

Доступ к сегментам TEB и PEB

Мы научились работать с памятью целевого процесса. Но есть одна проблема: доступ на чтение или запись конкретной переменной происходит по её абсолютному адресу. Вопрос в том, как его найти? Мы уже знаем, что его можно вычислить по базовому адресу сегмента, в котором находится эта переменная, и её смещению. Предположим, что мы знаем, какой сегмент следует искать. Как узнать его базовый адрес? К счастью, метаинформацию об адресном пространстве процесса можно найти в его памяти. Например, в специальных сегментах TEB и PEB.

В памяти процесса для каждого потока есть соответствующий ему TEB сегмент. Кроме прочей информации он содержит базовый адрес сегмента стека, выделенного этому потоку. В стеке же хранится большая часть переменных, используемых в потоке. Остальные переменные находятся в сегменте динамической памяти процесса, выделяемом по умолчанию. Его базовый адрес хранится в PEB сегменте. Следовательно, чтобы найти сегменты стека потока и динамической памяти процесса, нам надо найти PEB и соответствующий потоку TEB. Эта задача упрощается тем, что все TEB сегменты содержат базовый адрес PEB. Таким образом, задача сводится к поиску TEB сегмента.

Доступ к TEB текущего процесса

Главный поток 32-битного процесса

Рассмотрим методы доступа к TEB сегменту. Начнём с самого простого варианта этой задачи. Предположим, что у нас есть однопоточное приложение. Как ему получить доступ к TEB своего главного потока? Существует несколько способов.

Самый простой и прямолинейный метод – воспользоваться регистром FS процессора на x86 архитектуре или регистром GS на архитектуре x64. Вообще, процессор предоставляет ОС решать, как использовать эти регистры. Windows хранит в них указатель на TEB сегмент потока, который исполняется в данный момент. Листинг 3-3 демонстрирует чтение регистра FS.

Листинг 3-3. Функция `GetTeb`

```
#include <winternl.h>

PTEB GetTeb()
{
    PTEB pTeb;

    __asm {
        mov EAX, FS:[0x18]
        mov pTeb, EAX
    }
    return pTeb;
}
```

В функции `GetTeb` используются **ассемблерные вставки**. Эта возможность C++ позволяет добавлять в программу код на языке ассемблера, каждая команда которого соответствует одной инструкции процессора. Другими словами мы спускаемся на самый нижний уровень и оперируем элементарными действиями процессора.

Рассмотрим код `GetTeb` подробнее. Функция начинается с выделения памяти на стеке для локальной переменной `pTeb` типа `PTEB`. Согласно WinAPI документации, тип `PTEB` – это указатель на структуру, содержащую все данные сегмента TEB. Далее идёт блок с двумя командами на языке ассемблера:

1. Запись в регистр `EAX` некоторого значения. Оно находится по абсолютному адресу памяти, который рассчитывается по формуле:

```
1 линейный адрес = базовый адрес из регистра FS + 0x18
```

2. Запись значение регистра `EAX` в переменную `pTeb`.

В результате этих команд базовый адрес регистра TEB оказывается записан в переменную `pTeb`. Её мы и возвращаем из функции.

Почему `GetTeb` не может просто вернуть значение регистра FS? Ведь он, по идее, должен указывать на TEB сегмент. Чтобы ответить на этот вопрос, рассмотрим как в Windows происходит доступ к сегментам процесса.

Большинство современных ОС использует **защищённый режим процессора** (protected processor mode). В этом режиме **адресация сегментов** происходит через **глобальную таблицу дескрипторов** (Global Descriptor Table или GDT). В регистрах FS и GS хранится селектор, который является индексом записи в таблице дескрипторов. В этой записи находится базовый адрес сегмента TEB. Запрос к GDT по селектору выполняется аппаратным **блоком сегментации** (segmentation unit) процессора. Результат этого запроса временно хранится в процессоре и недоступен для приложений или ОС. Таким образом, у Windows нет эффективного способа узнать базовый адрес сегмента TEB. Его можно прочитать из таблицы дескрипторов через WinAPI-функции `GetThreadSelectorEntry` и `Wow64GetThreadSelectorEntry`, но этот способ неэффективен из-за накладных расходов. Именно поэтому в TEB сегменте хранится его собственный базовый адрес.

Если вы интересуетесь подробностями, пример использования функции `GetThreadSelectorEntry` приведён в следующем обсуждении на форуме.

Структура TEB определена в заголовочном файле `winternal.h`, который распространяется с Windows SDK. Она отличается для разных версий Windows. Поэтому важно, чтобы ваши версии ОС и Windows SDK совпадали. Перед началом работы с TEB структурой всегда уточняйте её поля в заголовочном файле.

Определение структуры TEB из Windows SDK версии 8.1 выглядит следующим образом:

```
1 typedef struct _TEB {
2     PVOID Reserved1[12];
3     PPEB ProcessEnvironmentBlock;
4     PVOID Reserved2[399];
```

```
    BYTE Reserved3[1952];
    PVOID TlsSlots[64];
    BYTE Reserved4[8];
    PVOID Reserved5[26];
    PVOID ReservedForOle;  // Windows 2000 only
    PVOID Reserved6[4];
    PVOID TlsExpansionSlots;
} TEB, *PTEB;
```

В ней среди прочих есть поле `ProcessEnvironmentBlock`, которое указывает на структуру PEB. Через него мы можем получить доступ к PEB сегменту.

Главный поток 64-битного процесса

Мы не можем просто заменить регистр FS на GS и использовать функцию `GetTeb` из листинга 3-3 на 64-разрядной системе. Проблема в том, что компилятор Visual Studio C++ не поддерживает ассемблерные вставки при компиляции 64-разрядных приложений. Вместо них следует использовать **встроенные функции компилятора** (compiler intrinsics).

Листинг 3-4 демонстрирует функцию `GetTeb`, переписанную для поддержки обеих архитектур: x86 и x64.

Листинг 3-4. Функция `GetTeb` для архитектур x86 и x64

```
#include <windows.h>
#include <winternl.h>

PTEB GetTeb()
{
#if defined(_M_X64) // x64
    PTEB pTeb = reinterpret_cast<PTEB>(__readgsqword(0x30));
#else // x86
    PTEB pTeb = reinterpret_cast<PTEB>(__readfsdword(0x18));
#endif
    return pTeb;
}
```

В новом варианте `GetTeb` используется директива условной компиляции **препроцессора**. С её помощью перед компиляцией выбирается подходящая реализация функции. Если макрос `_M_X64` определён, значит целевая архитектура приложения 64-разрядная. В этом случае вызывается встроенная функция компилятора `__readgsqword`, которая читает 64-битное значение со смещением 0x30 от базового адреса сегмента TEB (на него указывает регистр GS через селектор). Для 32-разрядной архитектуры вызывается встроенная функция `__readfsdword`, которая читает 32-битное значение со смещением 0x18 от базового адреса сегмента TEB (на него указывает регистр FS).

Новая реализация функции `GetTeb` может вызвать вопрос: почему поле структуры TEB с базовым адресом сегмента имеет разные смещения для x86 и x64 архитектур? Чтобы ответить на него, рассмотрим определение структуры `NT_TIB`, которая используется для представления части TEB, независимой от версии Windows:

```
typedef struct _NT_TIB {
    struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList;
```

```
    PVOID StackBase;
    PVOID StackLimit;
    PVOID SubSystemTib;
     union
     {
          PVOID FiberData;
          ULONG Version;
     };
    PVOID ArbitraryUserPointer;
    struct _NT_TIB *Self;
} NT_TIB;
```

Поле с базовым адресом сегмента TEB называется `Self`. До него идут шесть полей, каждое из которых имеет тип `PVOID`. `PVOID` – это указатель на область памяти. Его размер зависит от разрядности процессора: 32 бита (или 4 байта) для архитектуры x86 и 64 бита (или 8 байт) для x64. Таким образом, в первом случае поле `Self` окажется смещено на 24 байта (6 * 4), а во втором на 48 байт (6 * 8). Переведём эти числа в шестнадцатеричную систему счисления и получим 0x18 и 0x30 соответственно.

Вместо того чтобы указывать смещения явно, мы можем использовать информацию о них из структуры `NT_TIB`. Листинг 3-5 демонстрирует это решение.

Листинг 3-5. Универсальная версия функции `GetTeb`

```
#include <windows.h>
#include <winternl.h>

PTEB GetTeb()
{
#if defined(_M_X64) // x64
    PTEB pTeb = reinterpret_cast<PTEB>(__readgsqword(reinterpret_cast<DWORD>(
                                        &static_cast<PNT_TIB>(nullptr)->Self)));
#else // x86
    PTEB pTeb = reinterpret_cast<PTEB>(__readfsdword(reinterpret_cast<DWORD>(
                                        &static_cast<PNT_TIB>(nullptr)->Self)));
#endif
    return pTeb;
}
```

Эта реализация функции `GetTeb` заимствована из статьи. В ней используются уже знакомые нам встроенные функции компилятора `__readgsqword` и `__readfsdword`. Мы применяем определение структуры `NT_TIB`, чтобы прочитать смещение её поля `Self`, содержащее базовый адрес сегмента TEB. Для этого мы последовательно **приводим типы**. Общий алгоритм расчёта смещения выглядит следующим образом:

1. Указатель на нулевой абсолютный адрес, который обозначается литералом `nullptr`, приводим к типу `PNT_TIB` с помощью оператора `static_cast`. Таким образом мы получаем указатель на структуру типа `NT_TIB`, расположенную по адресу 0.
2. С помощью оператора доступа к полю `->` читаем поле `Self` структуры `NT_TIB`.
3. С помощью операции взятия адреса `&` читаем абсолютный адрес поля `Self`. В данном случае абсолютный адрес совпадёт с относительным, поскольку он считается от нуля.

4. Приведём полученный относительный адрес поля `Self` к типу `DWORD` или `QWORD` (в зависимости от целевой архитектуры) с помощью оператора `reinterpret_cast`. Это приведение необходимо, так как встроенные функции компилятора ожидают конкретный тип входного параметра.

Версия функции `GetTeb` из листинга 3-5 позволяет исключить явное указание смещений в коде. Благодаря этому она будет корректно работать для всех версий Windows даже в тех, где эти смещения изменятся.

WinAPI-функции доступа к TEB

Получить доступ к TEB сегменту можно и через WinAPI. Функция `NtCurrentTeb` реализует тот же алгоритм, что и `GetTeb` из листинга 3-5. С её помощью можно получить указатель на структуру типа `TEB` текущего потока. Листинг 3-6 демонстрирует использование `NtCurrentTeb`.

Листинг 3-6. Пример вызова WinAPI-функции `NtCurrentTeb`

```
#include <windows.h>
#include <winternl.h>

PTEB pTeb = NtCurrentTeb();
```

Теперь все манипуляции над регистрами FS и GS происходят на уровне системной библиотеки ОС. Мы можем рассчитывать на её корректную работу для всех архитектур, поддерживаемых Windows (x86, x64, ARM).

До сих пор мы рассматривали случай однопоточного приложения. Если например нам нужно получить TEB вспомогательного потока из функции `main` (то есть главного потока), то все рассмотренные выше способы не подходят.

WinAPI-функция `NtQueryInformationThread` предоставляет доступ к TEB любого потока. Она работает только в контексте вызывающего процесса, т.е. с её помощью вы не сможете прочитать TEB игрового приложения из бота. Но в некоторых случаях `NtQueryInformationThread` может быть полезна. Листинг 3-7 демонстрирует реализацию `GetTeb`, которая использует `NtQueryInformationThread`.

Листинг 3-7. Функция`GetTeb`, вызывающая `NtQueryInformationThread`

```
#include <windows.h>
#include <winternl.h>

#pragma comment(lib,"ntdll.lib")

typedef struct _CLIENT_ID {
    DWORD UniqueProcess;
    DWORD UniqueThread;
} CLIENT_ID, *PCLIENT_ID;

typedef struct _THREAD_BASIC_INFORMATION {
    typedef PVOID KPRIORITY;
    NTSTATUS ExitStatus;
    PVOID TebBaseAddress;
    CLIENT_ID ClientId;
    KAFFINITY AffinityMask;
    KPRIORITY Priority;
    KPRIORITY BasePriority;
```

```
} THREAD_BASIC_INFORMATION, *PTHREAD_BASIC_INFORMATION;

typedef enum _THREADINFOCLASS2 {
    ThreadBasicInformation,
    ThreadTimes,
    ThreadPriority,
    ThreadBasePriority,
    ThreadAffinityMask,
    ThreadImpersonationToken,
    ThreadDescriptorTableEntry,
    ThreadEnableAlignmentFaultFixup,
    ThreadEventPair_Reusable,
    ThreadQuerySetWin32StartAddress,
    ThreadZeroTlsCell,
    ThreadPerformanceCount,
    ThreadAmILastThread,
    ThreadIdealProcessor,
    ThreadPriorityBoost,
    ThreadSetTlsArrayAddress,
    _ThreadIsIoPending,
    ThreadHideFromDebugger,
    ThreadBreakOnTermination,
    MaxThreadInfoClass
} THREADINFOCLASS2;

PTEB GetTeb()
{
    THREAD_BASIC_INFORMATION threadInfo;
    if (NtQueryInformationThread(GetCurrentThread(),
                                 (THREADINFOCLASS)ThreadBasicInformation,
                                 &threadInfo, sizeof(threadInfo), NULL))
    {
        printf("NtQueryInformationThread return error\n");
        return NULL;
    }
    return reinterpret_cast<PTEB>(threadInfo.TebBaseAddress);
}
```

Параметры функции `NtQueryInformationThread` приведены в таблице 3-5.

Таблица 3-5. Параметры функции `NtQueryInformationThread`

Параметр	Описание
`GetCurrentThread()`	Дескриптор целевого потока, TEB которого требуется прочитать. В примере используется дескриптор текущего потока.
`ThreadBasicInformation`	Константа типа перечисление (enum) `THREADINFOCLASS`. Она определяет тип структуры, возвращаемой функцией.
`threadInfo`	Указатель на структуру, в которую функция запишет свой результат.
`sizeof(...)`	Размер структуры с результатом работы функции. В нашем случае – это размер `threadInfo`.
`NULL`	Указатель на переменную. В неё запишется итоговый размер структуры с результатом (`threadInfo`).

Чтобы прочитать структуру типа `THREAD_BASIC_INFORMATION` для заданного потока, мы должны передать в функцию `NtQueryInformationThread` константу `ThreadBasicInformation` из перечисления `THREADINFOCLASS`. К сожалению, эта

константа недокументированна. Кроме того, она не определена в заголовочном файле `winternl.h`. В нём есть только константа `ThreadIsIoPending`.

Чтобы использовать недокументированную константу, её надо определить самостоятельно. Для этого добавим новое перечисление типа `THREADINFOCLASS2`, которое содержит нужную нам `ThreadBasicInformation`. Подробнее об этой константе, вы можете узнать в [неофициальной документации].

В нашем новом перечислении `THREADINFOCLASS2` не должно быть константы с именем `ThreadIsIoPending`, иначе она будет конфликтовать с определением из заголовочного файла `winternl.h`. Поэтому в листинге 3-7 мы переименовали её на `_ThreadIsIoPending`.

Функция `NtQueryInformationThread` возвращает структуру данных, тип который зависит от переданного вторым параметром константы. Если мы передаём недокументированную константу `ThreadBasicInformation`, то тип возвращаемой структуры будет также недокументирован. Поэтому мы должны самостоятельно определить её тип `THREAD_BASIC_INFORMATION`. Вы можете найти его в уже упомянутой неофициальной документации или скопировать из листинга 3-7.

Обратите внимание на определение структуры `THREAD_BASIC_INFORMATION`. Базовый адрес сегмента TEB хранится в её поле `TebBaseAddress`. Она отличается от структуры `TEB`, с которой мы сталкивались ранее.

Функция `NtQueryInformationThread` доступна через Native API интерфейс. Она реализована в динамической библиотеке `ntdll.dll`, которая всегда входит в состав дистрибутива Windows. Эта библиотека активно используется системами ОС. Но, чтобы вызвать её функции из пользовательского приложения, понадобится библиотека импорта `ntdll.lib` и заголовочный файл `winternl.h`. Windows SDK предоставляет эти файлы.

Воспользоваться библиотекой импорта можно с помощью **директивы pragma**:

```
1 #pragma comment(lib, "ntdll.lib")
```

Эта строчка добавляет файл `ntdll.lib` в список библиотек импорта, которым воспользуется компоновщик.

В архиве примеров к этой книге вы можете найти файл `TebPebSelf.cpp`, в котором приведены все рассмотренные нами способы доступа к TEB и PEB сегментам.

Доступ к TEB целевого процесса

Мы рассмотрели случай, когда приложение получает доступ к своим TEB сегментам. Такая задача редко возникает на практике, потому что все переменные доступны по своим именам и их не нужно искать в сегментах стека и динамической

памяти. С другой стороны благодаря этой упрощённой задаче, мы разобрались в устройстве сегмента TEB.

Теперь перейдём к реальной практической задаче и рассмотрим методы доступа к сегментам TEB и PEB целевого процесса. В качестве цели воспользуемся любым стандартным Windows приложением.

Для тестирования дальнейших примеров необходимо выполнить следующие шаги:

1. Запустить стандартное Windows приложение (например Notepad). Помните, что его разрядность совпадает с разрядностью Windows.
2. Прочитайте PID процесса приложения с помощью Task Manager.
3. Присвойте прочитанный PID переменной `pid` функции `main` в коде соответствующего примера:

```
1 DWORD pid = 5356;
```

4. Скомпилируйте пример.
5. Запустите его из командной строки с правами администратора.

После выполнения приложение напечатает результат в командную строку.

Повторение базового адреса TEB

Начнём с простейшего случая, когда целевой процесс – это однопоточное приложение. При его старте ОС назначает базовый адрес TEB главного потока. Очень часто этот адрес оказывается одним и тем же для 32-разрядных приложений. Воспользуемся этим наблюдением и составим простой алгоритм для чтения TEB сегмента целевого процесса:

1. Прочитать базовый адрес TEB сегмента главного потока текущего процесса.
2. Прочитать сегмент по этому же базовому адресу в адресном пространстве целевого процесса.

Листинг 3-8 демонстрирует реализацию этого алгоритма.

Листинг 3-8. Приложение `TebPebMirror.cpp`

```
1 #include <windows.h>
2 #include <winternl.h>
3
4 BOOL SetPrivilege(HANDLE hToken, LPCTSTR lpszPrivilege, BOOL bEnablePrivilege)
5 {
6     // Смотрите реализацию этой функции в листинге 3-1
7 }
8
9 BOOL GetMainThreadTeb(DWORD dwPid, PTEB pTeb)
10 {
11     LPVOID tebAddress = NtCurrentTeb();
12     printf("TEB = %p\n", tebAddress);
13
14     HANDLE hProcess = OpenProcess(PROCESS_VM_READ, FALSE, dwPid);
15     if (hProcess == NULL)
16         return false;
```

```
17
18      if (ReadProcessMemory(hProcess, tebAddress, pTeb, sizeof(TEB), NULL) == FALSE)
19      {
20          CloseHandle(hProcess);
21          return false;
22      }
23
24      CloseHandle(hProcess);
25      return true;
26  }
27
28  int main()
29  {
30      HANDLE hProc = GetCurrentProcess();
31
32      HANDLE hToken = NULL;
33      if (!OpenProcessToken(hProc, TOKEN_ADJUST_PRIVILEGES, &hToken))
34          printf("Failed to open access token\n");
35
36      if (!SetPrivilege(hToken, SE_DEBUG_NAME, TRUE))
37          printf("Failed to set debug privilege\n");
38
39      DWORD pid = 7368;
40
41      TEB teb;
42      if (!GetMainThreadTeb(pid, &teb))
43          printf("Failed to get TEB\n");
44
45      printf("PEB = %p StackBase = %p\n", teb.ProcessEnvironmentBlock,
46             teb.Reserved1[1]);
47
48      return 0;
49  }
```

После запуска приложения `TebPebMirror.cpp`, в командной строке будут распечатаны базовые адреса трёх сегментов целевого процесса:

- TEB
- PEB
- Сегмент стека главного потока

Мы использовали уже знакомый нам метод предоставления привилегии `SE_DEBUG_NAME` для маркера доступа текущего процесса с помощью WinAPI-функций `OpenProcessToken` и `SetPrivilege`. После этого вызывается функция `GetMainThreadTeb`, которая принимает входным параметром PID целевого процесса и возвращает указатель на структуру `TEB`. Алгоритм `GetMainThreadTeb` следующий:

1. Прочитать базовый адрес TEB сегмента текущего потока с помощью вызова `NtCurrentTeb`.
2. Получить дескриптор целевого процесса с правами доступа `PROCESS_VM_READ`. Для этого используется WinAPI-функция `OpenProcess`.
3. Прочитать структуру `TEB` целевого процесса с помощью вызова `ReadProcessMemory`.

В общем случае, при старте нового процесса Windows назначает базовый адрес сегмента TEB произвольно. Для 32-разрядных приложений этот адрес часто оказывается одним и тем же. Но для 64-разрядных приложений, он меняется при

каждом запуске. Поэтому рассмотренный нами метод доступа к TEB не рекомендуется применять в реальных ботах. Благодаря своей простоте он хорош только в качестве обучающего примера.

Приложение из листинга 3-8 успешно справляется с однопоточными целевыми процессами. Может ли оно работать с многопоточными? Да, но для этого надо немного изменить его код. Приложение должно создавать столько же вспомогательных потоков, сколько имеет целевой процесс. Для каждого потока надо прочитать базовый адрес соответствующего TEB сегмента. Затем через эти адреса можно пытаться получить доступ к сегментам TEB целевого процесса.

Узнать число потоков в целевом процессе можно с помощью отладчика WinDbg или OllyDbg. Достаточно открыть его карту памяти и посчитать число TEB сегментов в ней.

Для всех примеров этой главы важно помнить, что разрядность целевого процесса и вашего приложения должна быть одинаковой. Чтобы выбрать разрядность компилируемого приложения в Visual Studio, укажите желаемую целевую архитектуру в элементе интерфейса “Solution Platforms” (платформы для решения).

Перебор всех потоков целевого процесса

Попробуем найти надёжный способ чтения TEB сегментов целевого процесса. Обратимся к WinAPI. Он предоставляет функции прохода по всем потокам, работающим на данный момент в ОС. С их помощью мы можем узнать дескрипторы потоков целевого процесса. Зная эти дескрипторы можно прочитать все TEB сегменты через уже знакомую нам функцию `NtQueryInformationThread`.

WinAPI-функции прохода по списку активных потоков следующие:

- `CreateToolhelp32Snapshot` делает снимок текущего состояния системы со всеми запущенными процессам, их потоками, модулями и сегментами динамической памяти. В функцию можно передать PID целевого процесса, тогда в снимок попадёт только он и его ресурсы.
- `Thread32First` начинает перебор потоков в указанном снимке состояния системы. Функция записывает результат своей работы в структуру типа `THREADENTRY32`, переданную входным параметром по указателю. Эта структура содержит информацию о первом потоке в снимке.
- `Thread32Next` продолжает перебор потоков в указанном снимке. Имеет те же входные и выходные параметры, что и функция `Thread32First`.

Приложение `TebPebTraverse.cpp` из листинга 3-9 демонстрирует алгоритм перебора потоков.

Листинг 3-9. Приложение `TebPebTraverse.cpp`

```
1 #include <windows.h>
2 #include <tlhelp32.h>
```

```
#include <winternl.h>

#pragma comment(lib,"ntdll.lib")

typedef struct _CLIENT_ID {
    // Смотрите определение этой структуры в листинге 3-7
} CLIENT_ID, *PCLIENT_ID;

typedef struct _THREAD_BASIC_INFORMATION {
    // Смотрите определение этой структуры в листинге 3-7
} THREAD_BASIC_INFORMATION, *PTHREAD_BASIC_INFORMATION;

typedef enum _THREADINFOCLASS2
{
    // Смотрите определение этой структуры в листинге 3-7
}   THREADINFOCLASS2;

PTEB GetTeb(HANDLE hThread)
{
    THREAD_BASIC_INFORMATION threadInfo;
    NTSTATUS result = NtQueryInformationThread(hThread,
                                    (THREADINFOCLASS)ThreadBasicInformation,
                                    &threadInfo, sizeof(threadInfo), NULL);
    if (result)
    {
        printf("NtQueryInformationThread return error: %d\n", result);
        return NULL;
    }
    return reinterpret_cast<PTEB>(threadInfo.TebBaseAddress);
}

void ListProcessThreads(DWORD dwOwnerPID)
{
    HANDLE hThreadSnap = INVALID_HANDLE_VALUE;
    THREADENTRY32 te32;

    hThreadSnap = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);

    if (hThreadSnap == INVALID_HANDLE_VALUE)
        return;

    te32.dwSize = sizeof(THREADENTRY32);

    if (!Thread32First(hThreadSnap, &te32))
    {
        CloseHandle(hThreadSnap);
        return;
    }

    DWORD result = 0;
    do
    {
        if (te32.th32OwnerProcessID == dwOwnerPID)
        {
            printf("\n     THREAD ID = 0x%08X", te32.th32ThreadID);

            HANDLE hThread = OpenThread(THREAD_ALL_ACCESS, FALSE,
                                        te32.th32ThreadID);
            PTEB pTeb = GetTeb(hThread);
            printf("\n     TEB = %p\n", pTeb);

            CloseHandle(hThread);
        }
    } while (Thread32Next(hThreadSnap, &te32));

    printf("\n");
    CloseHandle(hThreadSnap);
}

int main()
{
    DWORD pid = 4792;
```

```
75
76      ListProcessThreads(pid);
77
78      return 0;
79  }
```

Это приложение выводит в консоль список потоков целевого процесса. Для каждого из них указывается идентификатор, назначенный ОС (аналог PID для потока), и базовый адрес соответствующего TEB сегмента.

Вся работа приложения происходит в функции `ListProcessThreads`, в которую передаётся PID целевого процесса. Для создания снимка состояния системы и работы с ним привилегия `SE_DEBUG_NAME` не требуется. Поэтому при запуске примера будет достаточно предоставить ему только права администратора.

Алгоритм работы функции `ListProcessThreads` следующий:

1. Сделать снимок состояния системы через WinAPI вызов `CreateToolhelp32Snapshot`.
2. Начать проход по потокам в снимке с помощью функции `Thread32First`.
3. Сравнить PID процесса, которому принадлежит последний прочитанный поток, с PID целевого процесса.
4. Если идентификаторы совпадают, прочитать `TEB` структуру этого потока с помощью функции `GetTeb`.
5. Вывести в консоль полученную информацию о потоке.
6. Перейти к следующему потоку в снимке состояния системы через вызов `Thread32Next`. Повторить шаги 3, 4, 5 для каждого потока в снимке.

Метод доступа к TEB из листинга 3-9 надёжен и работает для многопоточных целевых процессов любой разрядности. Применяйте в своих приложениях именно его.

Может быть не совсем понятно, как различать потоки при переборе их функцией `Thread32Next`. Например, вы ищете TEB главного потока. Структура `THREADENTRY32` не содержит идентификатор потока в терминах процесса. Вместо этого в ней есть только глобальный ID, которым пользуется менеджер объектов Windows.

При использовании функции `Thread32Next` можно полагаться на порядок следования TEB сегментов в адресном пространстве процесса. Другими словами, TEB сегмент с наибольшим базовым адресом соответствует главному потоку (ID которого равен 0). Следующий за ним сегмент с меньшим адресом соответствует потоку с ID 1 в терминах процесса и т.д. Вы можете проверить порядок следования TEB сегментов с помощью отладчика WinDbg.

Доступ к динамической памяти

Мы рассмотрели метод чтения базового адреса сегмента динамической памяти по умолчанию из структуры PEB. Однако, у процесса может быть несколько таких сегментов. К ним можно получить доступ через WinAPI-функции. Они позволяют перебрать все сегменты динамической памяти указанного процесса. Алгоритм их использования очень похож на перебор активных потоков в снимке состояния системы.

Следующие WinAPI-функции позволяют получить доступ к сегментам динамической памяти:

- `CreateToolhelp32Snapshot` уже знакомая нам функция, которая создаёт снимок текущего состояния системы.
- `Heap32ListFirst` начинает перебор сегментов динамической памяти, попавших в указанный снимок. Результат работы функции сохраняется в структуре типа `HEAPLIST32`.
- `Heap32ListNext` продолжает перебор сегментов в снимке. Имеет те же входные и выходные параметры, что и функция `Heap32ListFirst`.

WinaAPI также предоставляет две функции для перебора блоков сегментов динамической памяти: `Heap32First` и `Heap32Next`. Мы не будем их использовать в примерах этой главы.

Перебор блоков сегментов динамической памяти требует значительного времени, если целевой процесс представляет собой большое и сложное приложение.

Листинг 3-10 демонстрирует перебор сегментов динамической памяти целевого процесса.

Листинг 3-10. Приложение `HeapTraverse.cpp`

```
#include <windows.h>
#include <tlhelp32.h>

void ListProcessHeaps(DWORD pid)
{
    HEAPLIST32 hl;

    HANDLE hHeapSnap = CreateToolhelp32Snapshot(TH32CS_SNAPHEAPLIST, pid);

    hl.dwSize = sizeof(HEAPLIST32);

    if (hHeapSnap == INVALID_HANDLE_VALUE)
    {
        printf("CreateToolhelp32Snapshot failed (%d)\n", GetLastError());
        return;
    }

    if (Heap32ListFirst(hHeapSnap, &hl))
    {
        do
        {
            printf("\nHeap ID: 0x%lx\n", hl.th32HeapID);
```

```
23              printf("\Flags: 0x%lx\n", hl.dwFlags);
24          } while (Heap32ListNext(hHeapSnap, &hl));
25      }
26      else
27          printf("Cannot list first heap (%d)\n", GetLastError());
28
29      CloseHandle(hHeapSnap);
30  }
31
32  int main()
33  {
34      DWORD pid = 6712;
35
36      ListProcessHeaps(pid);
37
38      return 0;
39  }
```

Это приложение выводит в консоль базовый адрес и флаги каждого сегмента динамической памяти целевого процесса. ID каждого сегмента соответствует его базовому адресу. Флаги важны, поскольку позволяют отличать сегменты друг от друга. Например, сегмент динамической памяти по умолчанию всегда имеет ненулевые флаги.

Функция `ListProcessHeaps` очень похожа по принципу работы на `ListProcessThreads` из листинга 3-9. Её алгоритм выглядит следующим образом:

1. Сделать снимок состояния системы с ресурсами только целевого процесса через вызов `CreateToolhelp32Snapshot`.
2. Начать проход по сегментам динамической памяти в снимке с помощью функции `Heap32ListFirst`.
3. Вывести в консоль ID и флаги текущего сегмента.
4. Повторить шаг 3 для всех сегментов в снимке, которые перебираются функцией `Heap32ListNext`.

Сегменты динамической памяти перебираются в порядке увеличения их ID. Сегмент с меньшим ID будет пройден раньше, чем сегмент с большим. Эта информация может помочь, когда вам понадобится отличить один сегмент от другого.

Выводы

Мы рассмотрели методы чтения базовых адресов сегментов стека и динамической памяти, которые могут содержать состояние игровых объектов. Любой внутриигровой бот использует их в том или ином виде для доступа к памяти игрового приложения.

Пример бота для Diablo 2

Обзор игры Diablo 2

Мы узнали достаточно, чтобы написать простого внутриигрового бота. Он будет автоматизировать некоторые действия в известной RPG Diablo 2. Её игровой процесс типичен для жанра: игрок должен выполнять квесты, убивать монстров и развивать своего персонажа.

Наш бот будет следить за состоянием игрового персонажа. Как только один из его параметров (например здоровье) опустится ниже порогового значения, бот будет выполнять некоторое действие (например использовать зелье лечения).

Перед тем как начать писать код, познакомимся с интерфейсом игры. Скриншот окна Diablo 2 приведён на иллюстрации 3-15. В центре находится игровой персонаж. Слева и справа от него – монстры, один из которых выделен курсором мыши. В нижней части окна находится панель управления. На ней есть четыре слота с зельями лечения, которые привязаны к горячим клавишам. Наш бот будет использовать предметы в этих слотах по мере необходимости.

Иллюстрация 3-15. Скриншот окна Diablo 2

Все параметры персонажа приведены на иллюстрации 3-16. На ней вы видите два открытых внутриигровых окна: левое и правое. В верхней части левого находится общая информация о персонаже: имя Kain, класс Paladin, уровень 70, очки опыта 285160782. Ниже указаны параметры персонажа, влияющие на игровую механику. Например, “Strength” (сила) определяет урон, наносимый противнику при ударе.

Иллюстрация 3-16. Параметры игрового персонажа

Правое окно на иллюстрации 3-16 отображает дерево способностей персонажа. Способности позволяют наносить больше урона противникам. Каждая из них имеет уровень, который определяет её эффективность. Более подробная информация о параметрах и способностях персонажа доступна на Wiki.

В Diablo 2 есть два режима игры: однопользовательский и многопользовательский. Мы будем рассматривать только однопользовательский. В нём вы сможете останавливать игру под отладчиком в любой момент на неограниченное время, чтобы исследовать адресное пространство её процесса. В многопользовательском режиме этому будут мешать тайм-ауты. Если игровой клиент не отвечает какое-то время, сервер его отключает.

Чтобы протестировать нашего бота, вы можете купить игру Diablo 2 на официальном сайте разработчика. Альтернативное решение – воспользоваться бесплатным клоном игры под названием Flaire. В этом случае вам придётся немного изменить код бота самостоятельно. Diablo 2 отличается от своего клона интерфейсом и сложностью. Память процесса оригинальной игры намного сложнее анализировать из-за большого количество вспомогательных библиотек.

Задачи бота

Прежде всего, чётко определим наши цели. Мы не собираемся взламывать игру, то есть нарушать её правила и вмешиваться в механику игрового процесса. Примеры подобных взломов вы можете найти в статьях Jan Miller:

- extreme-gamerz.org/diablo2/viewdiablo2/hackingdiablo2

- www.battleforums.com/threads/howtohackd2-edition-2.111214

Наш бот следует правилам игры. Он реагирует на изменение состояния персонажа и симулирует действие. При этом параметры всех игровых объектов меняются согласно правилам. Процесс Diablo 2 продолжает работать по своим оригинальным алгоритмам так же, как если бы действия совершал игрок.

Мы рассмотрели параметры персонажа. Из них проще всего контролировать уровень здоровья. Он уменьшается, когда игрок получает урон от монстров. При использовании зелья лечения – увеличивается. Учитывая эту механику, наш бот может работать по следующему алгоритму:

1. Прочитать текущий уровень здоровья игрового персонажа.
2. Сравнить этот уровень с пороговым значением.
3. Если здоровье меньше порога, использовать зелье лечения.

Этот алгоритм позволит игровому персонажу выживать до тех пор, пока у него остаются зелья лечения. Однако, несмотря на кажущуюся простоту, для реализации бота нам придётся хорошо разобраться в структуре памяти процесса Diablo 2.

Исследование памяти процесса Diablo 2

Мы готовы приступить к исследованию памяти процесса Diablo 2. Наша задача – найти переменную, которая хранит значение текущего здоровья персонажа.

Выполним предварительную настройку окна Diablo 2, чтобы с ним было удобнее работать. Сразу после установки игра запускается в полноэкранном режиме. Это неудобно, если приходится часто переключаться на отладчик или сканер памяти.

Чтобы запустить игру в оконном режиме, выполните следующие действия:

1. Щёлкните правой кнопкой мыши по иконке “Diablo II” на рабочем столе. В открывшемся меню выберите пункт “Properties” (свойства).
2. В диалоге “Properties” перейдите на вкладку “Shortcut” (ярлык).
3. В поле “Target” (объект) добавить параметр “-w”. В результате полная команда запуска приложения будет выглядеть так:

```
1 "C:\DiabloII\Diablo II.exe" -w
```

Если вы запустите Diablo 2 через настроенную иконку на рабочем столе, приложение откроется в оконном режиме. Чтобы начать игру, нажмите кнопку “Single player” (одиночная игра) в главном меню и создайте нового персонажа.

Поиск параметров персонажа

Найдём уровень здоровья игрового персонажа в памяти процесса Diablo 2. Для этого воспользуемся сканером Cheat Engine. Он разработан именно для решения

подобных задач.

Если вы попробуете найти уровень здоровья по его текущему значению без предварительной настройки Cheat Engine, поиск не даст результата. Вероятнее всего, после первого сканирования вы получите длинный список предполагаемых адресов. При повторном поиске (кнопка “Next Scan”) после изменения уровня здоровья персонажа, список результатов станет пустым.

Прямолинейный подход не заработал. Это совершенно нормально для больших и сложных приложений, как Diablo 2. В памяти процесса находится очень много игровых объектов, причём параметры некоторых из них совпадают. Мы не знаем, как именно они хранятся в памяти. Поэтому будет разумно сначала разобраться с этим вопросом. Если мы сможем найти нужный нам объект в памяти, получить доступ к его параметрам будет очень просто.

Ещё раз обратимся к окну с параметрами игрового персонажа. Значения некоторых из них наверняка уникальны и не встречаются у других игровых объектов. Какие именно? Возможны следующие варианты:

1. **Имя персонажа**
Очень маловероятно, что есть объект с тем же именем, которое игрок дал своему персонажу. Если это всё-таки произошло, всегда можно создать нового персонажа с другим уникальным именем.

2. **Очки опыта**
Это длинное положительное целочисленное число. Число такого размера может встретиться в другом объекте только случайно. Если Cheat Engine всё же нашёл несколько потенциальных адресов, очки опыта персонажа очень просто увеличить. Убейте одного-двух монстров и выполните повторное сканирование памяти кнопкой “Next Scan”.

3. **Значение выносливости**
Это ещё одно длинное число, которое определяет, как долго игрок способен быстро двигаться по карте. Его очень просто уменьшить: для этого достаточно перемещать персонажа вне города.

Из всех вариантов, предлагаю искать очки опыта персонажа. Если вы только начали игру, вам нужно убить нескольких монстров, чтобы этот параметр стал больше нуля. Иллюстрация 3-17 демонстрирует окно Cheat Engine с возможным результатом поиска. Сканер нашёл несколько переменных с одинаковым значением. Только некоторые из них относятся к объекту игрового персонажа. Другие могут быть связаны с интерфейсом игры и выводом информации на экран.

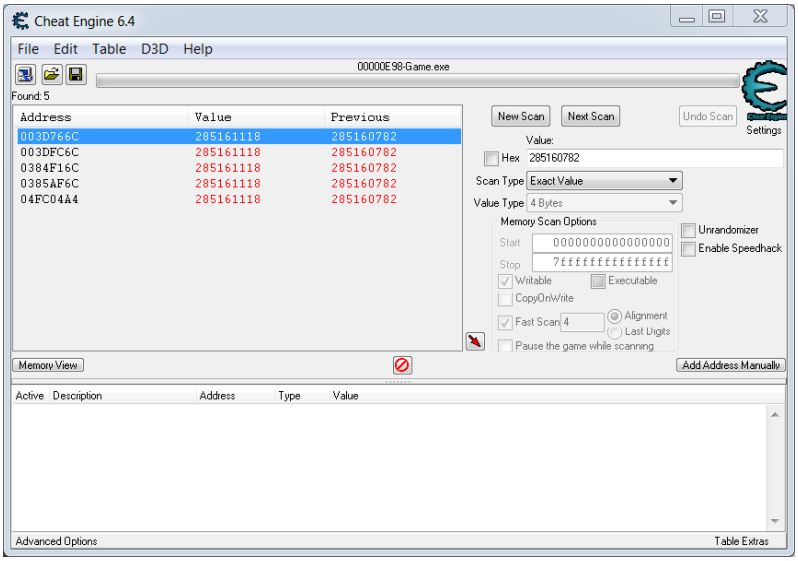


Иллюстрация 3-17. Результаты поиска параметра игрового персонажа

Теперь определим, какие из найденных параметров относятся к объекту персонажа. Тип сегмента, в котором они хранятся, может дать нам подсказку.

Запустите отладчик WinDbg, подключитесь к работающему процессу Diablo 2 и выполните команду `!address`. Сегменты с найденными параметрами выглядят следующим образом:

```
1 + 0`003c0000  0`003e0000  0`00020000  MEM_PRIVATE MEM_COMMIT PAGE_READWRITE <unknown>
2 + 0`03840000  0`03850000  0`00010000  MEM_PRIVATE MEM_COMMIT PAGE_READWRITE <unknown>
3 + 0`03850000  0`03860000  0`00010000  MEM_PRIVATE MEM_COMMIT PAGE_READWRITE <unknown>
4 + 0`04f50000  0`04fd0000  0`00080000  MEM_PRIVATE MEM_COMMIT PAGE_READWRITE <unknown>
```

Отладчик не смог определить тип этих сегментов и указал, что он неизвестен (“unknown”). Мы знаем, что WinDbg умеет корректно определять сегменты стека и динамической памяти. Если тип неизвестен, скорее всего, это не первое и не второе.

Сегменты неизвестного типа может выделять WinAPI-функция `VirtualAllocEx`. Чтобы это проверить, воспользуемся простым тестовым приложением. Файл `VirtualAllocEx.cpp` с его исходным кодом есть в архиве с примерами для этой книги. Если вы запустите приложение под отладчиком WinDbg и прочитаете его адресное пространство, вы увидите один сегмент с неизвестным типом. Функция `VirtualAllocEx` выделяет его и возвращает базовый адрес.

Вернёмся к процессу Diablo 2. Все сегменты, хранящие переменные со значением очков опыта персонажа, имеют одинаковый тип. Следовательно, мы не сможем их отличить по этому признаку. Это важно, поскольку после перезапуска игры, порядок следования сегментов может измениться. Если мы не сможем их отличить, мы не определим сегмент, в котором находится игровой объект персонажа. Размер

сегмента тоже не подходит в качестве критерия проверки, потому что он совпадает у двух сегментов.

Попробуем другой подход. Очевидно, что параметры персонажа меняются, когда игрок совершает действия. Например, после любого перемещения персонажа по карте, его координата изменится. Мы можем следить за такими изменениями в области памяти около найденных нами адресов параметра очков опыта. У Cheat Engine есть возможность отображения области памяти в реальном времени. Чтобы ею воспользоваться, надо открыть окно Memory Viewer (просмотрщик памяти). Для этого выполните следующие шаги:

1. Выберите один из адресов в списке результатов поиска.
2. Щёлкните по нему правой кнопкой мыши.
3. Выберите пункт “Browse this memory region” (просмотреть эту область памяти) в открывшемся меню.

Откроется окно Memory Viewer, как показано на иллюстрации 3-18. Оно разделено на две части. В верхней части выводится область памяти около выбранного адреса в виде **дизассемблированного кода**. Это значит, что Cheat Engine пытается представить данные в виде инструкций процессора. В нижней части окна отображаются данные той же самой области памяти в шестнадцатеричном формате. Обе части окна Memory Viewer выводят одни и те же данные, но представленные в разном виде.

Нас интересует нижняя половина окна. Данные, соответствующие очкам опыта персонажа, подчёркнуты красным на иллюстрации 3-18. В моём примере персонаж имеет 285161118 очков опыта.

Почему последовательность байт “9E 36 FF 10” равна числу 285161118? Мы запускаем Diablo 2 на процессоре с архитектурой x86, которая имеет **порядок байт** от младшего к старшему (little-endian byte order). Следовательно, значение из окна Memory Viewer нужно перевернуть, чтобы получить правильно число. Другими словами, последовательность байтов “9E 36 FF 10” надо интерпретировать как “10 FF 36 9E”. Вы можете воспользоваться стандартным приложением Windows Calculator, чтобы перевести число 10FF369E в десятичную систему и получить 285161118.

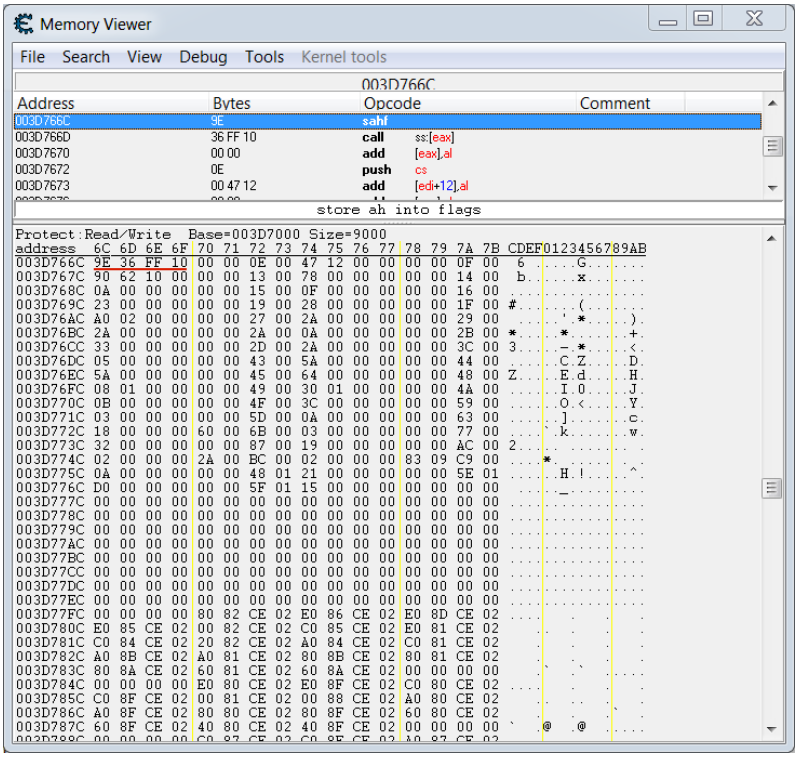


Иллюстрация 3-18. Окно Memory Viewer сканера Cheat Engine

Окно Memory Viewer позволяет настроить формат вывода данных. Для этого щёлкните правой кнопкой мыши в любом месте нижней половины окна и выберите пункт “Display Type” (тип отображения) в открывшемся меню. Дальше вы можете выбрать нужный вам тип. Однако, я рекомендую всегда пользоваться форматом “Byte hex”, как на иллюстрации 3-18. Другие форматы могут вызвать путаницу, потому что объединяют соседние байты в числа. Когда размер искомых чисел неизвестен, их фрагменты могут объединяться неправильно.

Теперь попробуем проследить изменения данных в областях памяти. Для удобства разместите окна Memory Viewer и Diablo 2 рядом, но без перекрытия, как изображено на иллюстрации 3-19. Это позволит вам одновременно управлять персонажем и следить за изменениями в памяти.

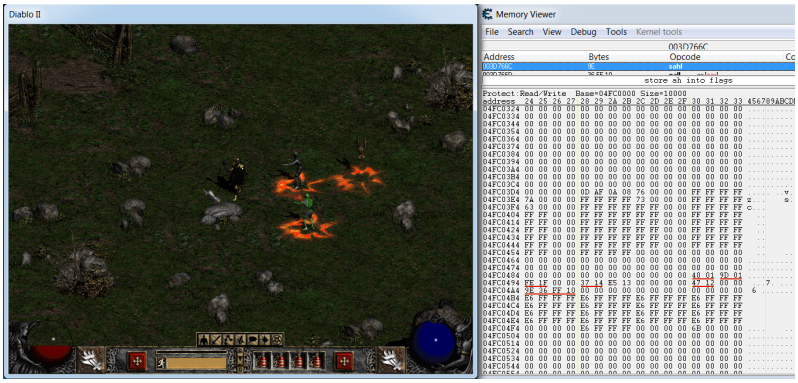


Иллюстрация 3-19. Исследование изменений в памяти процесса Diablo 2

В окне Memory Viewer, приведённом на иллюстрации 3-19, открыта область памяти около адреса 04FC04A4. Это один из адресов, который мы получили при поиске очков опыта персонажа. Вам нужно исследовать области около каждого из них.

Как мы поймём, что нашли объект игрового персонажа в памяти? Предлагаю простое правило: если объект хранит больше параметров персонажа чем другие, то его информация наиболее полная и боту следует использовать именно его. В моём случае этот объект имеет адрес 04FC04A4 и находится последним в списке результатов сканирования Cheat Engine.

Таблица 3-6 демонстрирует параметры, которые мы обнаружили в объекте.

Таблица 3-6. Найденные параметры игрового объекта

Параметр	Адрес	Смещение	Размер	Шестнадцатеричное значение	Десятичное значение
Здоровье	04FC0490	490	2	40 01	320
Мана	04FC0492	492	2	9D 01	413
Выносливость	04FC0494	494	2	FE 1F	8190
Координата X	04FC0498	498	2	37 14	5175
Координата Y	04FC04A0	4A0	2	47 12	4679
Очки опыта	04FC04A4	4A4	4	9E 36 FF 10	285161118

Эти параметры подчёркнуты красным на иллюстрации 3-19. Чтобы их обнаружить, я выполнял следующие игровые действия:

1. Оставаться на месте и получать урон от атакующего монстра. В этом случае уменьшается только параметр здоровья по адресу 04FC0490.
2. Оставаться на месте и использовать любую способность. В этом случае уменьшается запас маны персонажа. Соответствующая переменная находится по адресу 04FC0492.

3. Перемещаться бегом вне города. При этом действии меняются сразу три параметра: выносливость, координаты X и Y. Если персонаж бегает достаточно долго, его выносливость уменьшится до нуля. Тогда можно отличить в памяти её значение (по адресу 04FC0494) от координат. Если перемещать персонажа только в горизонтальном или вертикальном направлении будет меняться одна из координат (X по адресу 04FC0498 или Y по 04FC04A0).
4. Убить любого монстра. В результате увеличатся очки опыта персонажа. Адрес соответствующей переменой равен 04FC04A4. Этот параметр легко отличить от уровней здоровья и маны, поскольку они наоборот обычно уменьшаются во время сражения с монстрами.

Что мы узнали нового о параметрах персонажа? Во-первых, уровень здоровья хранится в двухбайтовой переменной. Следовательно, чтобы найти его в памяти, надо указать “2 Byte” в поле “Value Type” (тип значения) окна Cheat Engine перед поиском.

Также мы выяснили, что у некоторых параметров нет четырехбайтового выравнивания. Это означает, что их адреса не кратны четырём. Например, уровень маны по адресу 04FC0492. Чтобы найти значения таких параметров, вам надо убрать галочку “Fast Scan” (быстрое сканирование) в окне Cheat Engine.

Правильная конфигурация Cheat Engine для поиска параметров игрового персонажа приведена на иллюстрации 3-20. Красным подчёркнуты изменённые настройки.

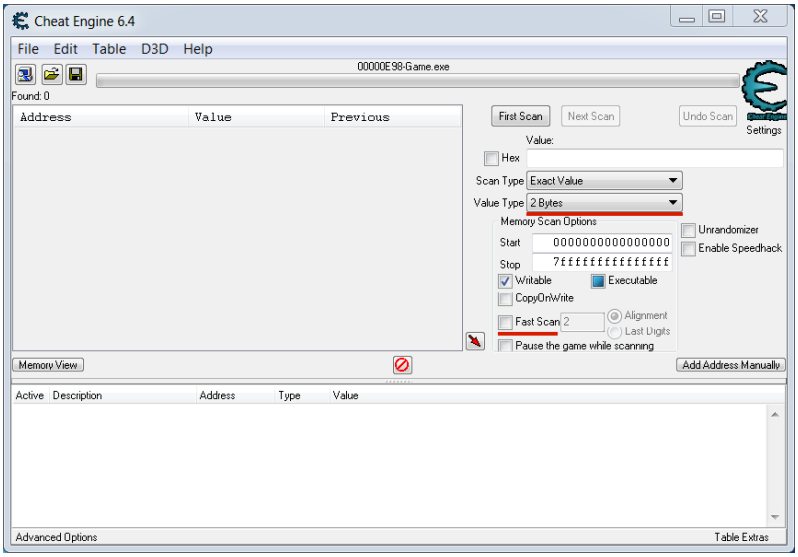


Иллюстрация 3-20. Конфигурация Cheat Engine

Возможно, вы обратили внимание на столбец “Смещение” в таблице 3-6. В нём указаны смещения каждого параметра относительно адреса начала объекта. Рассмотрим, как найти этот адрес в памяти процесса.

Поиск объекта в памяти

Задумаемся над тем, как наш бот будет искать параметр здоровья персонажа в памяти процесса Diablo 2. Эту задачу можно разделить на два этапа:

1. Найти объект персонажа.
2. Добавить к адресу объекта постоянное смещение, чтобы получить адрес параметра.

Можем ли мы быть уверены, что смещение параметра будет всегда постоянным? Если приложение написано на C++ или C (обычно именно эти языки применяют для разработки игр), параметры игрового объекта, скорее всего, будут храниться в структуре или классе (особый вид структуры). Структура – это тип, в котором все поля и их порядок жёстко определены. Поэтому при каждом запуске приложения смещение полей структуры от её начала остаётся неизменным.

Мы знаем, как искать игровой объект в памяти приложения с помощью Cheat Engine. К сожалению, наш бот не может пользоваться сканером памяти. Точнее такое решение было бы слишком громоздким. Вместо этого, он должен полагаться на собственные алгоритмы. Поэтому нам нужно найти способ поиска объекта на единственном снимке памяти, который доступен боту через WinAPI-функции.

Прокрутите окно Memory Viewer вверх от переменной с очками опыта в сторону младших адресов. Вы обнаружите имя персонажа, как на иллюстрации 3-21. Четыре байта, подчёркнутых красным, представляют собой строку “Kain”. Обратите внимание, что порядок байтов для строк не перевернут на процессорах с little-endian архитектурой. Причина в том, что внутренняя структура ASCII-строк и массивов с элементами в один байт совпадает. Процессор обрабатывает байтовые массивы поэлементно, то есть читает в свои регистры по одному байту и никаких перестановок не происходит.

Ещё раз посмотрите на иллюстрацию 3-21. Легко заметить, что область памяти в сторону младших адресов от имени персонажа занулена. Предположим, что это признак границы игрового объекта. Можем ли мы проверить эту гипотезу?

Воспользуемся OllyDbg, чтобы поставить точку останова (breakpoint) на адрес переменной с именем персонажа. Когда какой-то код процесса Diablo 2 попытается прочитать или записать значение по этому адресу, процесс остановится и отладчик получит управление. Мы сможем проанализировать этот код и, возможно, найдём признаки начала игрового объекта.

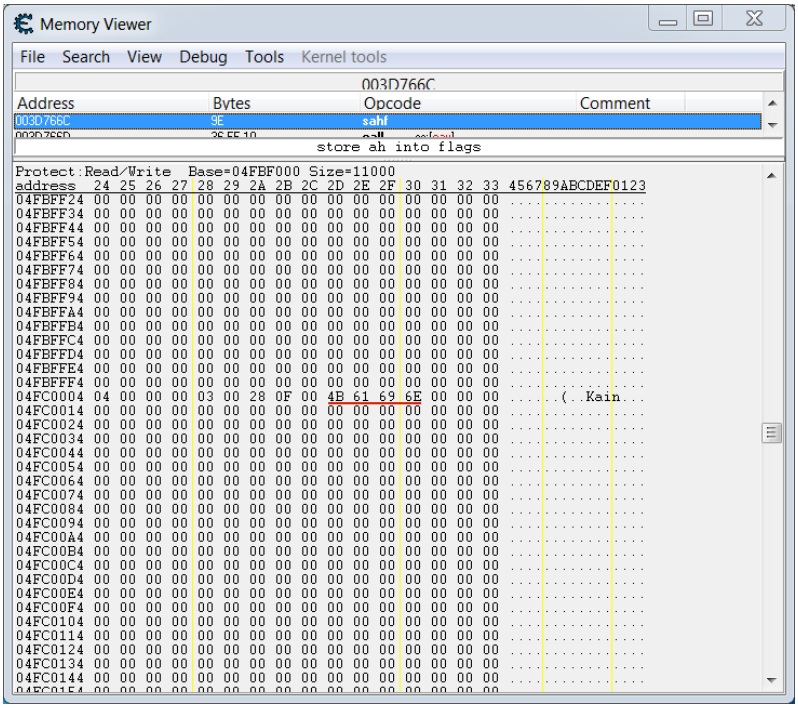


Иллюстрация 3-21. Имя персонажа в памяти процесса

Алгоритм поиска границ объекта с помощью отладчика OllyDbg выглядит следующим образом:

1. Запустите отладчик с правами администратора и подключитесь к уже запущенному процессу Diablo 2.
2. Щёлкните правой кнопкой мыши в левом нижнем окне OllyDbg и переключитесь на шестнадцатеричный формат **дампа памяти**.
3. Нажмите комбинацию клавиш Ctrl+G, чтобы открыть диалог “Enter expression to follow” (ввести выражение для перехода) для поиска адреса в памяти.
4. Введите адрес строки с именем персонажа в поле “Enter address expression” (ввести адрес выражения) диалога поиска. В моём случае это адрес 04FC000D. Нажмите кнопку “Follow expression” (перейти к выражению). Теперь курсор в окне с дампом памяти указывает на первый байт строки.
5. Прокрутите окно дампа памяти вверх, чтобы найти первый ненулевой байт, с которого предположительно начинается объект персонажа. Выделите этот байт щелчком левой кнопки мыши.
6. Нажмите комбинацию клавиш Shift+F3, чтобы открыть диалог “Set memory breakpoint” для установки точки останова. Выберите в диалоге галочки “Read access” (доступ на чтение) и “Write access” (доступ на запись), чтобы точка останова срабатывала на чтение и запись по выбранному адресу памяти. Нажмите кнопку “OK”.
7. Нажмите F9, чтобы продолжить выполнение процесса Diablo 2. Он остановится несколько раз. Продолжайте его выполнение по нажатию F9, пока процесс не

будет стабильно работать. В этом случае вы увидите состояние “Running” в правом нижнем углу окна отладчика.

8. Переключитесь на окно Diablo 2. Сразу после этого сработает наша точка останова.
9. Переключитесь на окно OllyDbg. Оно должно выглядеть так же, как на иллюстрации 3-22.

Дизассемблированный код процесса отображается в левом верхнем окне отладчика. Инструкция процессора с адресом 03668D9F, исполнение которой вызвало срабатывание нашей точки останова, выделена серой линией:

```
1 CMP DWORD PTR DS:[ESI+4], 4
```

Эта инструкция сравнивает константу 4 и число типа DWORD, хранящееся по адресу “ESI + 4”. Регистр **ESI** используется для указания на источник данных в инструкциях процессора. Регистр **DS** хранит базовый адрес сегмента с данными. Как правило, регистры ESI и DS используются совместно. В правом верхнем окне отладчика отображается текущее значение всех регистров процессора. ESI хранит адрес 04FC0000.

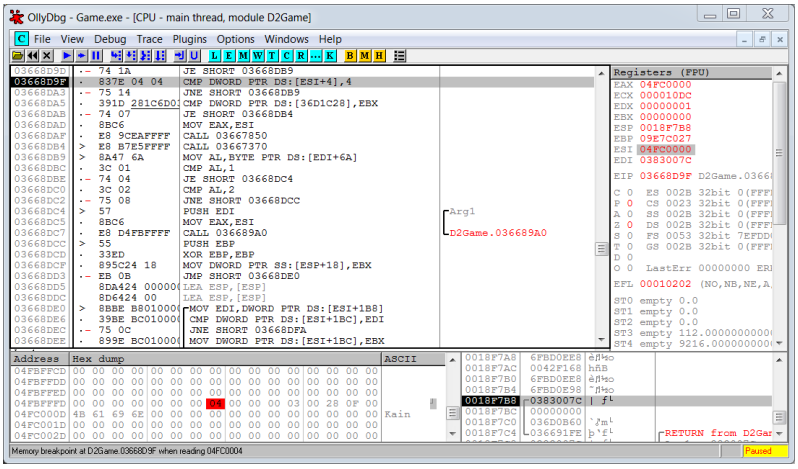


Иллюстрация 3-22. Точка останова на начале объекта игрового персонажа

Изучим дизассемблированный код после инструкции, на которой сработала точка останова. На иллюстрации 3-22 найдите следующий код, начинающийся по адресу 03668DE0:

```
1 MOV EDI,DWORD PTR DS:[ESI+1B8]
2 CMP DWORD PTR DS:[ESI+1BC],EDI
3 JNE SHORT 03668DFA
4 MOV DWORD PTR DS:[ESI+1BC],EBX
```

Эти инструкции выглядят как обращения к полям структуры в C++ или C. Константы 1B8 и 1BC – это смещения полей от её начала. Если вы прокрутите дизассемблированный код ниже, вы найдёте ещё несколько подобных обращений. Следовательно, адрес начала структуры, в которой хранятся параметры игрового персонажа, равен 04FC0000, то есть текущему значению регистра ESI.

Теперь мы можем вычислить смещение параметра здоровья от начала структуры:

```
1 04FC0490 - 04FC0000 = 0x490
```

Смещение равно 490 в шестнадцатеричной системе счисления.

Следующий вопрос: как бот найдёт адрес начала объекта игрового персонажа в памяти? Мы знаем, что этот объект хранится в сегменте неизвестного (unknown) типа, размер которого 80000 байт в шестнадцатеричной системе. У сегмента есть три флага: MEM_PRIVATE, MEM_COMMIT и PAGE_READWRITE. В адресном пространстве процесса Diablo 2 есть минимум десять сегментов этого же типа, размера и с теми же флагами. Следовательно, мы не можем просто перебрать все сегменты и найти нужный по этим признакам.

Ещё раз рассмотрим первые несколько байт объекта персонажа:

```
1 00 00 00 00 04 00 00 00 03 00 28 0F 00 4B 61 69 6E 00 00 00
```

Если перезапустить игру и найти объект снова, эти байты будут теми же. Можно предположить, что эта последовательность байтов представляет собой неизменяемые параметры персонажа. Они задаются однократно при его создании и больше никогда не меняются.

Список неизменяемых параметров персонажа следующий:

- Имя.
- Флаг, означающий что персонаж играет в расширенную версию Diablo 2.
- Флаг hardcore режима. Он означает, что игра закончится после первой смерти персонажа.
- Класс персонажа.

Последовательность неизменных байтов в начале объекта можно использовать как цель для поиска. Назовём её **магическим числом** или сигнатурой Учтите, что в вашем случае эта последовательность будет отличаться.

Проверим предположение о неизменных параметрах с помощью Cheat Engine. Запустите сканер и подключитесь к процессу Diablo 2. Выберите пункт “Array of byte” (массив байт) в поле “Value Type”. Затем выберите галочку “Hex” и скопируйте свою последовательность байт в поле “Array of byte”. Ожидаемый результат поиска представлен на иллюстрации 3-23.

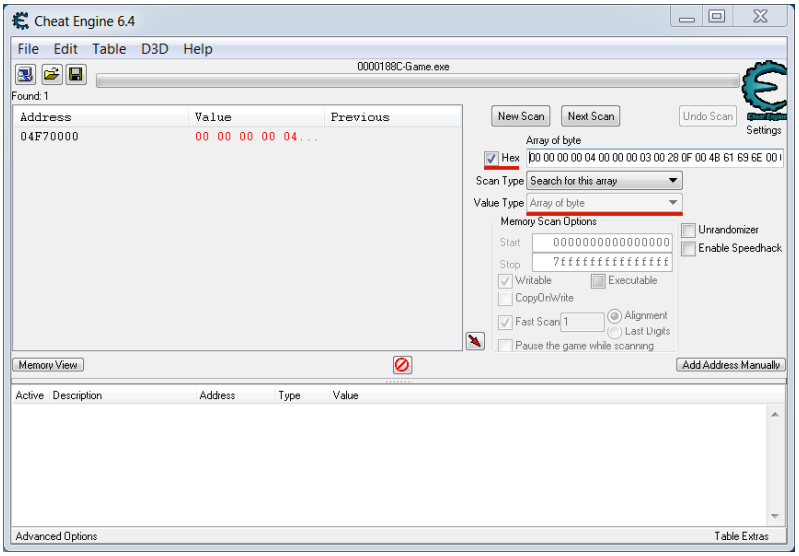


Иллюстрация 3-23. Поиск объекта игрового персонажа в памяти процесса Diablo 2

Если вы перезапустите игру, адрес объекта изменится. На иллюстрации 3-23 он равен 04F70000. Тем не менее, смещения всех параметров персонажа внутри объекта остаются неизменными. Исходя из этого, абсолютный адрес уровня здоровья персонажа в нашем случае будет равен 04F70490, т.к. его смещение равно 490.

Есть альтернативный способ найти уровень здоровья персонажа с помощью Cheat Engine. Он может быть полезен при первоначальном анализе памяти игрового приложения. Cheat Engine предоставляет функцию сканирования указателей (pointer scanning). С её помощью можно найти базовый адрес и смещение переменной после нескольких этапов сканирования памяти процесса. К сожалению, в некоторых случаях эта функция не работает. Подробнее о ней можно узнать в статье.

Реализация бота

Мы собрали всю необходимую информацию, чтобы реализовать нашего внутриигрового бота. Составим подробный алгоритм его работы:

1. Предоставить привилегию `SE_DEBUG_NAME` процессу бота.
2. Подключиться к процессу Diablo 2 для доступа к его памяти.
3. Искать объект игрового персонажа в адресном пространстве игры.
4. Вычислить абсолютный адрес параметра здоровья персонажа.
5. Читать значение параметра в бесконечном цикле. Как только оно опустится ниже 100 пунктов, использовать зелье лечения.

Мы уже рассмотрели реализацию первого шага алгоритма в предыдущем разделе этой главы.

Второй шаг алгоритма можно реализовать двумя способами:

1. Указать PID целевого процесса в коде бота, как мы делали в предыдущих примерах.
2. Определять PID динамически по активному в данный момент окну.

Во втором случае важно следить, чтобы в момент запуска бота было активно именно окно Diablo 2. Благодаря этому подходу им будет намного удобнее пользоваться, поскольку его не придётся перекомпилировать с корректным PID целевого процесса перед каждым запуском.

Листинг 3-11 демонстрирует чтение PID и подключение к процессу Diablo 2.

Листинг 3-11. Код подключения к процессу

```
int main()
{
    Sleep(4000);

    HWND wnd = GetForegroundWindow();
    DWORD pid = 0;
    if (!GetWindowThreadProcessId(wnd, &pid))
    {
        printf("Error of the pid detection\n");
        return 1;
    }

    HANDLE hTargetProc = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
    if (!hTargetProc)
    {
        printf("Failed to open process: %u\n", GetLastError());
    }
    return 0;
}
```

Перед началом работы мы ждём четыре секунды с помощью WinAPI-функции `Sleep`. Этого времени должно быть достаточно, чтобы вы успели переключиться на окно Diablo 2.

Для чтения PID процесса мы использовали две новые WinAPI-функции:

1. `GetForegroundWindow` возвращает дескриптор активного в данный момент окна.
2. `GetWindowThreadProcessId` возвращает PID процесса, который владеет окном, указанным по его дескриптору.

Прочитанный PID активного окна сохраняется в переменную `pid`.

Третий шаг алгоритма заключается в поиске объекта игрового персонажа в памяти процесса. Для этого предлагаю воспользоваться подходом, описанном в серии видеоуроков. В них рассматривается разработка простого сканера памяти, алгоритм работы которого очень похож на Cheat Engine. Идея заключается в переборе всех сегментов процесса Diablo 2 с помощью WinAPI-функции `VirtualQueryEx`.

Код для поиска объекта персонажа в памяти процесса приведён в листинге 3-12.

Листинг 3-12. Код поиска игрового объекта в памяти процесса

```
SIZE_T IsArrayMatch(HANDLE proc, SIZE_T address, SIZE_T segmentSize,
                    BYTE array[], SIZE_T arraySize)
{
    BYTE* procArray = new BYTE[segmentSize];

    if (ReadProcessMemory(proc, (void*)address, procArray, segmentSize, NULL) != 0)
    {
        printf("Failed to read memory: %u\n", GetLastError());
        delete[] procArray;
        return 0;
    }

    for (SIZE_T i = 0; i < segmentSize; ++i)
    {
        if ((array[0] == procArray[i]) && ((i + arraySize) < segmentSize))
        {
            if (!memcmp(array, procArray + i, arraySize))
            {
                delete[] procArray;
                return address + i;
            }
        }
    }

    delete[] procArray;
    return 0;
}

SIZE_T ScanSegments(HANDLE proc, BYTE array[], SIZE_T size)
{
    MEMORY_BASIC_INFORMATION meminfo;
    LPCVOID addr = 0;
    SIZE_T result = 0;

    if (!proc)
        return 0;

    while (1)
    {
        if (VirtualQueryEx(proc, addr, &meminfo, sizeof(meminfo)) == 0)
            break;

        if ((meminfo.State & MEM_COMMIT) && (meminfo.Type & MEM_PRIVATE)
            && (meminfo.Protect & PAGE_READWRITE)
            && !(meminfo.Protect & PAGE_GUARD))
        {
            result = IsArrayMatch(proc, (SIZE_T)meminfo.BaseAddress,
                                  meminfo.RegionSize, array, size);

            if (result != 0)
                return result;
        }
        addr = (unsigned char*)meminfo.BaseAddress + meminfo.RegionSize;
    }
    return 0;
}

int main()
{
    // Предоставить SE_DEBUG_NAME привилегию текущему процессу

    // Подключиться к процессу Diablo 2

    BYTE array[] = { 0, 0, 0, 0, 0x04, 0, 0, 0, 0x03, 0, 0x28,
                     0x0F, 0, 0x4B, 0x61, 0x69, 0x6E, 0, 0, 0 };

    SIZE_T objectAddress = ScanSegments(hTargetProc, array, sizeof(array));

    return 0;
}
```

Алгоритм прохода по сегментам памяти целевого процесса реализован в функции `ScanSegments`. Она возвращает указатель на объект персонажа и принимает на вход три параметра:

1. Дескриптор процесса Diablo 2.
2. Указатель на искомую последовательность байт.
3. Размер последовательности.

Алгоритм `ScanSegments` состоит из следующих шагов:

1. Прочитать сегмент памяти с базовым адресом равным переменной `addr` с помощью функции `VirtualQueryEx`.
2. Проверить совпадают ли флаги прочитанного сегмента с флагами искомого. Если нет, перейти к следующему сегменту.
3. Искать последовательность байт, характерную для объекта персонажа в прочитанном сегменте.
4. Если последовательность найдена, вернуть её абсолютный адрес. Иначе читать следующий сегмент.

Алгоритм поиска последовательности байт в сегменте реализован в функции `IsArrayMatch`. Он выглядит следующим образом:

1. Прочитать все данные из указанного сегмента с помощью WinAPI-функции `ReadProcessMemory`.
2. Искать в этих данных последовательность путём побайтного сравнения.

Не забывайте, что искомая последовательность байт отличается в вашем случае.

Четвёртый шаг общего алгоритма бота – это вычисление абсолютного адреса параметра здоровья персонажа. Для этого воспользуемся переменной `objectAddress`, хранящей результат вызова функции `ScanSegments`. Прибавим к ней смещение параметра в объекте по следующей формуле:

```
1 SIZE_T hpAddress = objectAddress + 0x490;
```

Теперь абсолютный адрес, по которому можно прочитать параметр здоровья, находится в переменной `hpAddress`.

Последним действием бот проверяет уровень здоровья персонажа. Если он оказался ниже порогового значения, бот должен использовать зелье лечения. Реализация этой проверки приведена в листинге 3-13.

Листинг 3-13. Код проверки уровня здоровья персонажа

```
WORD ReadWord(HANDLE hProc, DWORD_PTR address)
{
    WORD result = 0;

    if (ReadProcessMemory(hProc, (void*)address, &result, sizeof(result), NULL) == 0)
        printf("Failed to read memory: %u\n", GetLastError());

    return result;
}

int main()
{
    // Предоставить SE_DEBUG_NAME привилегию текущему процессу

    // Подключиться к процессу Diablo 2

    // Искать объект игрового персонажа в памяти процесса Diablo 2

    // Вычислить абсолютный адрес переменной с уровнем здоровья персонажа

    ULONG hp = 0;

    while (1)
    {
        hp = ReadWord(hTargetProc, hpAddress);
        printf("HP = %lu\n", hp);

        if (hp < 100)
            PostMessage(wnd, WM_KEYDOWN, 0x31, 0x1);

        Sleep(2000);
    }
    return 0;
}
```

Здоровье персонажа читается в бесконечном `while` цикле с помощью функции `ReadWord`, которая представляет собой обёртку для WinAPI вызова `ReadProcessMemory`. Прочитав значение здоровья, бот выводит его на консоль. Это позволит вам проверить, что параметр найден правильно. Сравните его значение с тем, что выводится в окне Diablo 2. Если уровень здоровья окажется меньше 100, бот симулирует нажатие горячей клавиши “1”. По нему игровой персонаж использует зелье лечения. Для симуляции нажатия клавиши вызывается WinAPI-функция `PostMessage`.

Вы можете возразить, что использование функции `PostMessage` – это не встраивание данных в память процесса, характерное для внутриигровых ботов. Вместо модификации памяти, мы внедряем сообщение `WM_KEYDOWN`, которое соответствует нажатию клавиши, в очередь сообщений процесса Diablo 2. Мы используем этот способ симуляции действий игрока для упрощения кода нашего примера. Более сложный подход рассматривается далее.

Параметры функции `PostMessage` описаны в таблице 3-7.

Таблица 3-7. Параметры функции `PostMessage`

Параметр	Описание
`wnd`	Дескриптор окна. Создавший это окно процесс получит сообщение.
`WM_KEYDOWN`	Код сообщения.
`0x31`	**Виртуальный код** нажатой клавиши.
`0x1`	Параметры нажатия. Самый важный из них – число срабатываний нажатия (хранится в битах с 0 по 15).

Симуляция нажатия клавиши не сработает, если четвёртый параметр функции `PostMessage` равен нулю.

Полная реализация бота доступна в файле `AutohpBot.cpp` из архива примеров к этой книге.

Для тестирования бота выполните следующие действия:

1. Измените последовательность байт для поиска так, чтобы она соответствовала вашему персонажу. В исходном коде бота это строка:

```
1 BYTE array[] = { 0, 0, 0, 0, 0x04, 0, 0, 0, 0x03, 0, 0x28, 0x0F, 0, 0x4B, 0x61, 0x69\
2 , 0x6E, 0, 0, 0 };
```

2. Скомпилируйте бота с новой последовательностью байт.
3. Запустите Diablo 2 в оконном режиме.
4. Запустите бота с правами администратора.
5. В течение четырёх секунд после старта бота переключитесь на окно Diablo 2. После этой задержки, бот подключится к процессу игры и начнёт следить за уровнем здоровья персонажа.
6. Найдите в игре монстра и получите от него урон так, чтобы здоровье персонажа опустилось ниже 100 пунктов.

В результате бот симулирует нажатие горячей клавиши “1”.

Не забудьте привязать к панели горячих клавиш зелье лечения. Для вызова справки по интерфейсу игры, нажмите клавишу H. Панель “Belt” (пояс) горячих клавиш находится в правой нижней части экрана. Вы можете перенести на неё зелья лечения левой кнопкой мыши.

Дальнейшие улучшения

Есть несколько изменений, которые могут значительно улучшить нашего бота. Рассмотрим их подробнее.

Главная проблема бота в том, что он нажимает только одну горячую клавишу из четырёх доступных. Из-за этого персонаж не будет использовать все зелья лечения, которые у него есть. Чтобы исправить это, перепишем цикл проверки параметра здоровья, как предлагается в листинге 3-14.

Листинг 3-14. Использование всех слотов панели горячих клавиш

```
    ULONG hp = 0;
    BYTE keys[] = { 0x31, 0x32, 0x33, 0x34 };
    BYTE keyIndex = 0;

    while (1)
    {
        hp = ReadWord(hTargetProc, hpAddress);
        printf("HP = %lu\n", hp);

        if (hp < 100)
        {
            PostMessage(wnd, WM_KEYDOWN, keys[keyIndex], 0x1);
            ++keyIndex;
            if (keyIndex == sizeof(keys))
                keyIndex = 0;
        }
        Sleep(2000);
    }
```

Теперь мы храним список горячих клавиш в байтовом массиве `keys`. Для его индексации используется переменная `keyIndex`. Она инкрементируется после каждого применения зелья лечения. При достижении конца массива, `keyIndex` сбрасывается в ноль. Таким образом бот будет использовать все слоты панели горячих клавиш. Когда зелья лечения в первом ряду панели закончатся, бот перейдёт ко второму ряду и т.д.

Бота можно улучшить, если мы добавим функцию контроля за уровнем маны персонажа. Для этого подойдёт такой же алгоритм, как и для проверки здоровья. Чтобы восстанавливать ману, бот может использовать специальное зелье.

Сейчас бот симулирует нажатие клавиши с помощью функции `PostMessage`. Вместо этого он может писать новое значение здоровья персонажа прямо в память процесса Diablo 2. Листинг 3-15 демонстрирует соответствующий код.

Листинг 3-15. Запись нового значения параметра персонажа в память процесса

```
void WriteWord(HANDLE hProc, DWORD_PTR address, WORD value)
{
    if (WriteProcessMemory(hProc, (void*)address, &value, sizeof(value), NULL) == 0)
        printf("Failed to write memory: %u\n", GetLastError());
}

int main()
{
    // Предоставить SE_DEBUG_NAME привилегию текущему процессу

    // Подключиться к процессу Diablo 2

    // Искать объект игрового персонажа в памяти процесса Diablo 2

    // Вычислить абсолютный адрес переменной с уровнем здоровья персонажа

    ULONG hp = 0;

```

```
19      while (1)
20      {
21          hp = ReadWord(hTargetProc, hpAddress);
22          printf("HP = %lu\n", hp);
23
24          if (hp < 100)
25              WriteWord(hTargetProc, hpAddress, 100);
26
27          Sleep(2000);
28      }
29      return 0;
30  }
```

Запись нового значения параметра персонажа происходит через WinAPI-функцию `WriteProcessMemory`. Для удобства работы с ней используется обёртка `WriteWord`. Теперь если уровень здоровья персонажа становится меньше 100, бот переписывает его значением 100 в памяти процесса. У этого подхода есть один серьёзный недостаток – он нарушает игровую механику. Параметр объекта меняется в обход алгоритмов игры. По этой причине состояние объекта может стать неконсистентным.

Попробуйте протестировать версию бота из листинга 3-15. В большинстве случаев уровень здоровья персонажа не будет меняться после записи нового значения. Причина в том, что приложение хранит этот параметр в нескольких местах (не только в объекте персонажа, который мы нашли). После записи ботом нового значения, у приложения есть несколько несовпадающих переменных для одного и того же параметра. Очевидно, что механика игры не может корректно обработать эту ситуацию и происходят ошибки. Запись данных в память процесса работает только в простых играх без многочисленных копий параметров объектов.

Есть ещё один способ встраивания данных в память процесса игрового приложения. Он основан на техниках внедрения кода, описанных в следующих статьях:

- www.codeproject.com/Articles/4610/Three-Ways-to-Inject-Your-Code-into-Another-Proces
- www.codeproject.com/Articles/9229/RemoteLib-DLL-Injection-for-Win-x-NT-Platforms

Идея заключается в том, чтобы заставить игровое приложение исполнять код бота в своём адресном пространстве. Если это удастся, бот сможет вызывать любую функцию игры или её библиотек. В этом случае не нужно симулировать нажатие клавиши. Можно просто напрямую вызвать функцию самой игры типа “UseHealPotion” (использовать зелье лечения). Однако, внедрение кода требует глубокого анализа и реверс-инжиниринга целевого приложения.

Алгоритм нашего бота очень простой. Он автоматизирует использование зелий лечения, и игрок может на них не отвлекаться. Можно ли написать более сложного бота, который бы самостоятельно убивал монстров? Эта задача выполнима. Самым

трудным шагом для бота будет поиск объектов монстров в памяти игрового процесса. Рассмотрим возможное решение.

Мы знаем, как и где хранятся координаты X и Y игрового персонажа (см. таблицу 3-6). Это два двухбайтовых числа, следующие друг за другом в памяти. Скорее всего, координаты других игровых объектов хранятся в таком же формате.

Теперь предположим, что когда монстр атакует персонажа, они находятся рядом друг с другом и их координаты отличаются незначительно. Бот мог бы сканировать память игрового процесса на наличие следующих друг за другом двухбайтовых чисел, значения которых близки к текущим координатам игрового персонажа. Многие результаты такого поиска будут ложными и их надо отфильтровать. Подсказкой для алгоритма фильтрации может быть то, что координаты всех видимых на экране монстров должны находиться в одном и том же сегменте памяти.

Бот может запомнить сегмент в котором хранятся найденные координаты монстров, а после этого искать их только в нём. Для атаки монстров бот может симулировать действия клавиатуры или мыши с помощью WinAPI-функции `PostMessage`.

Выводы

Мы реализовали простого внутриигрового бота для Diablo 2. Он использует характерные для своего типа техники взаимодействия с игрой. Рассмотрим его достоинства и недостатки. В принципе, мы можем обобщить их на любого внутриигрового бота.

Преимущества:

1. Бот получает точную информацию о состоянии игровых объектов. Ошибки и неточности как в кликерах крайне маловероятны.
2. Есть несколько способов встраивать действия бота в процесс игрового приложения: симулировать действия игрока, писать значения в память процесса, вызывать внутренние функции игры. Можно выбрать наиболее подходящий вариант.
3. Бот способен очень быстро реагировать на события в игре. Зачастую скорость его реакции выше, чем у игрока.

Недостатки:

1. Анализ памяти игрового процесса и его дизассемблированного кода требует значительных усилий и времени.
2. В большинстве случаев бот совместим только с одной версией игры, для которой он разрабатывался. Для новых версий его необходимо адаптировать.
3. Существует много эффективных средств защиты как от реверс-инжиниринга и отладки, так и от несанкционированного доступа к памяти процесса.

Основной недостаток внутриигровых ботов – это сложность их разработки и сопровождения. Но с другой стороны они очень надёжны в работе.

Методы защиты от внутриигровых ботов

Мы познакомились с принципами работы внутриигровых ботов. Теперь рассмотрим способы защиты от них. Есть две группы методов защиты:

- Защита приложения от реверс-инжиниринга.
- Блокировка алгоритмов бота.

Первая группа методов разрабатывается очень давно: со времён первых версий коммерческого ПО, которое нужно было защищать от нелицензионного распространения. Эти методы хорошо известны, и информацию о них легко найти в Интернете. Их основная задача – усложнить анализ приложения с помощью отладчика и дизассемблера.

Вторая группа методов защищает данные процесса игрового приложения от чтения и записи. Из-за них боту становится сложнее читать состояние объектов и внедрять свои действия.

Некоторые методы защиты можно отнести сразу к обеим группам.

Тестовое приложение

Вспомним архитектуру клиент-сервер современных онлайн-игр. Клиент выполняется на компьютере пользователя и обменивается сообщениями с игровым сервером. Большая часть методов защиты от внутриигровых ботов работает на стороне клиента.

Рассмотрим методы защиты на конкретном примере. Напишем простое приложение, которое будет имитировать игру и менять состояние некоторого объекта. Для контроля за этим состоянием напишем простейшего внутриигрового бота.

Все примеры этого раздела компилировались на Visual Studio C++ под 32 разрядную архитектуру. Некоторые из них могут не заработать или потребуют изменений, если вы соберёте их под 64 разрядную архитектуру или с помощью MinGW.

Алгоритм тестового приложения будет следующим:

1. При старте присвоить параметру объекта (например его уровень здоровья) максимально допустимое значение.
2. В цикле проверять состояние горячей клавиши “1”.

3. Если пользователь не нажимает клавишу, уменьшать параметр объекта. Иначе – увеличивать.
4. Если параметр оказался равен 0, завершить приложение.

Листинг 3-16 демонстрирует исходный код тестового приложения.

Листинг 3-16. Исходный код тестового приложения

```
#include <stdio.h>
#include <stdint.h>
#include <windows.h>

static const uint16_t MAX_LIFE = 20;
static uint16_t gLife = MAX_LIFE;

int main()
{
    SHORT result = 0;

    while (gLife > 0)
    {
        result = GetAsyncKeyState(0x31);
        if (result != 0xFFFF8001)
            --gLife;
        else
            ++gLife;

        printf("life = %u\n", gLife);
        Sleep(1000);
    }
    printf("stop\n");
    return 0;
}
```

Уровень здоровья игрового объекта хранится в глобальной переменной `gLife`. При старте приложения мы присваиваем ей значение константы `MAX_LIFE`, равное 20.

Вся работа функции `main` происходит в цикле `while`. В нём мы проверяем состояние клавиши “1” с помощью WinAPI-функции `GetAsyncKeyState`. Виртуальный код этой клавиши (равный 0x31) передаётся в функцию входным параметром. Если вызов `GetAsyncKeyState` возвращает состояние “не нажато”, переменная `gLife` уменьшается на единицу. В противном случае – увеличивается также на единицу. После этого идёт односекундная задержка для того, чтобы пользователь успел отпустить клавишу.

Попробуйте скомпилировать тестовое приложение в конфигурации “Debug” (отладка) в Visual Studio и запустить его.

Исследование памяти тестового приложения

Теперь напишем бота для нашего тестового приложения. Его алгоритм будет таким же, как и для игры Diablo 2 из прошлого раздела. Если параметр здоровья опускается ниже 10, бот симулирует нажатие клавиши “1”.

Чтобы контролировать параметр здоровья, бот должен читать значение переменной `gLife`. Очевидно, мы не можем воспользоваться тем же механизмом поиска объекта,

который мы применили для Diablo 2. Нам нужно проанализировать адресное пространство тестового приложения и найти подходящий метод доступа к `gLife`. Хорошая новость заключается в том, что это приложение очень простое и для его изучения нам будет достаточно отладчика OllyDbg.

Чтобы найти сегмент, содержащий переменную `gLife` выполните следующие шаги:

1. Запустите отладчик OllyDbg. Нажмите F3, чтобы открыть диалог “Select 32-bit executable” (выберите 32-разрядный исполняемый файл). В диалоге выберите скомпилированное приложение из листинга 3-16. В результате отладчик запустит приложение и остановит его процесс на первой исполняемой инструкции процессора.
2. Нажмите комбинацию клавиш Ctrl+G, чтобы открыть диалог “Enter expression to follow” (ввести выражение для перехода).
3. Введите имена EXE модуля и функции `main` через точку в поле диалога “Enter address expression” (ввести адрес выражения). Должна получиться строка “TestApplication.main”. После этого нажмите кнопку “Follow expression” (перейти к выражению). Теперь курсор окна дизассемблера должен указывать на первую инструкцию функции `main`.
4. Поставьте точку останова на эту инструкцию нажатием F2.
5. Начните исполнение процесса нажатием F9. Должна сработать наша точка останова.
6. Щёлкните правой кнопкой мыши по следующей строке дизассемблированного кода:

```
1 MOV AX,WORD PTR DS:[gLife]
```

Позиция курсора должна совпадать с иллюстрацией 3-24.

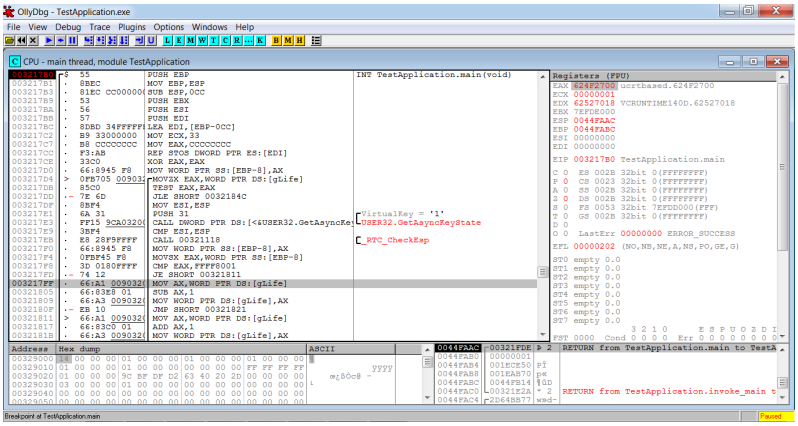
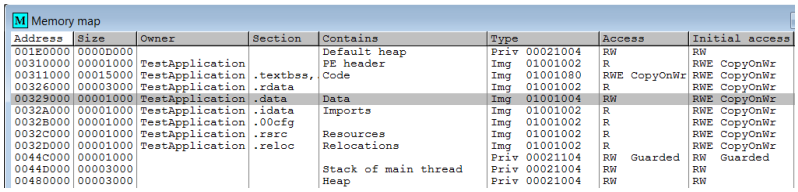


Иллюстрация 3-24. Точка останова в `main` функции

7. Выберите пункт “Follow in Dump” -> “Memory address” (“Следить в дампе” -> “Адрес памяти”) в открывшемся меню. Теперь курсор в окне дампа памяти указывает на переменную `gLife`. В моём случае она находится по адресу 329000 и имеет значение 14 в шестнадцатеричной системе.
8. Нажмите комбинацию клавиш Alt+M, чтобы открыть окно “Memory map” (карта памяти).
9. Найдите сегмент, в котором находится переменная `gLife`. Им окажется `.data` модуля TestApplication, как на иллюстрации 3-25.


Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial access
001E0000	0000D000			Default heap	Priv 00021004	RW	RW
00310000	00001000	TestApplication		PE header	Img 01001002	R	RWE CopyOnWr
00311000	00015000	TestApplication	.textbss,	Code	Img 01001080	RWE CopyOnWr	RWE CopyOnWr
00326000	00003000	TestApplication	.rdata		Img 01001002	R	RWE CopyOnWr
00329000	00001000	TestApplication	.data	Data	Img 01001004	RW	RWE CopyOnWr
0032A000	00001000	TestApplication	.idata	Imports	Img 01001002	R	RWE CopyOnWr
0032B000	00001000	TestApplication	.00cfg		Img 01001002	R	RWE CopyOnWr
0032C000	00001000	TestApplication	.rsrc	Resources	Img 01001002	R	RWE CopyOnWr
0032D000	00001000	TestApplication	.reloc	Relocations	Img 01001002	R	RWE CopyOnWr
0044C000	00001000				Priv 00021104	RW Guarded	RW Guarded
0044D000	00003000			Stack of main thread	Priv 00021004	RW	RW
00480000	00003000			Heap	Priv 00021004	RW	RW

Иллюстрация 3-25. Сегменты модуля TestApplication

Мы выяснили, что переменная `gLife` хранится в самом начале сегмента `.data`. Следовательно, её адрес равен базовому адресу сегмента. Если бот найдёт `.data`, он сразу сможет прочитать `gLife`.

Бот для тестового приложения

Мы рассмотрели алгоритм бота для тестового приложения в общих чертах. Теперь составим точную последовательность действий, которую затем запрограммируем:

1. Предоставить привилегию `SE_DEBUG_NAME` процессу бота.
2. Подключиться к процессу тестового приложения.
3. Искать в памяти сегмент `.data`, в котором хранится переменная `gLife`.
4. Читать переменную в бесконечном цикле. Если её значение оказывается меньше 10, записать вместо него 20.

Исходный код бота приведён в листинге 3-17.

Листинг 3-17. Исходный код бота для тестового приложения

```
#include <stdio.h>
#include <windows.h>

BOOL SetPrivilege(HANDLE hToken, LPCTSTR lpszPrivilege, BOOL bEnablePrivilege)
{
    // См. реализацию этой функции в листинге 3-1
}

SIZE_T ScanSegments(HANDLE proc)
{
    MEMORY_BASIC_INFORMATION meminfo;
    LPCVOID addr = 0;

    if (!proc)
        return 0;

    while (1)
    {
```

```
        if (VirtualQueryEx(proc, addr, &meminfo, sizeof(meminfo)) == 0)
            break;

        if ((meminfo.State == MEM_COMMIT) && (meminfo.Type & MEM_IMAGE)
            && (meminfo.Protect == PAGE_READWRITE)
            && (meminfo.RegionSize == 0x1000))
        {
            return (SIZE_T)meminfo.BaseAddress;
        }
        addr = (unsigned char*)meminfo.BaseAddress + meminfo.RegionSize;
    }
    return 0;
}

WORD ReadWord(HANDLE hProc, DWORD_PTR address)
{
    // См. реализацию этой функции в листинге 3-13
}

void WriteWord(HANDLE hProc, DWORD_PTR address, WORD value)
{
    if (WriteProcessMemory(hProc, (void*)address, &value, sizeof(value), NULL) == 0)
        printf("Failed to write memory: %u\n", GetLastError());
}

int main()
{
    // Предоставить SE_DEBUG_NAME привилегию текущему процессу

    // Подключиться к процессу тестового приложения

    SIZE_T lifeAddress = ScanSegments(hTargetProc);

    ULONG hp = 0;
    while (1)
    {
        hp = ReadWord(hTargetProc, lifeAddress);
        printf("life = %lu\n", hp);

        if (hp < 10)
            WriteWord(hTargetProc, lifeAddress, 20);

        Sleep(1000);
    }
    return 0;
}
```

Главное различие ботов для тестового приложения и для Diablo 2 – это реализация функции `ScanSegments`. Теперь мы можем отличить нужный нам сегмент `.data` по его флагам и размеру. Эта информация выводится в окне “Memory map” отладчика OllyDbg. Таблица 3-8 поясняет значения флагов.

Таблица 3-8. Значения флагов сегмента `.data`

Столбец окна “Memory map”	Значение в OllyDbg	Значение в WinAPI	Описание
Type	Img	MEM_IMAGE	Страницы памяти были загружены из исполняемого файла.
Access	RW	PAGE_READWRITE	Страницы памяти доступны для чтения и записи.
		MEM_COMMIT	Страницы памяти были выделены на физическом носителе: RAM или файл подкачки на жёстком диске.

Флаг `MEM_COMMIT` не отображается в OllyDbg, но его можно прочитать с помощью WinDbg.

Чтобы запустить бота, выполните следующие действия:

1. Запустите тестовое приложение.
2. Запустите бота с правами администратора.
3. Переключитесь на консоль с работающим тестовым приложением.
4. Ждите, пока не увидите сообщение, что переменная `gLife` стала меньше 10.

Бот перепишет значение `gLife`, как только оно станет слишко мало.

Защита приложения от реверс-инжиниринга

Сначала рассмотрим методы защиты кода и памяти игрового приложения от исследования. Как показал пример разработки бота для Diablo 2, знание внутренних аспектов работы игры очень важно. К сожалению, абсолютно надёжной защиты не бывает. Лучшее, чего можно достигнуть – заставить потенциального разработчика бота потратить больше времени на исследование игры. Возможно, этого будет достаточно, чтобы он отказался от своих планов.

WinAPI-функции для обнаружения отладчика

Основной инструмент для исследования памяти процесса – это отладчик. Поэтому самым прямолинейным способом защиты будет его обнаружение. Для этого WinAPI интерфейс предоставляет несколько подходящих функций. При обнаружении отладчика, достаточно будет просто завершить работу приложения.

Рассматриваемые далее методы не защищают память процесса от чтения сканером (например Cheat Engine) или ботом. Они только позволяют обнаружить факт подключения отладчика.

IsDebuggerPresent

WinAPI-функция `IsDebuggerPresent` возвращает значение `true`, если к вызвавшему её процессу подключён отладчик. `IsDebuggerPresent` можно использовать следующим образом:

```
int main()
{
    if (IsDebuggerPresent())
    {
        printf("debugger detected!\n");
        exit(EXIT_FAILURE);
    }

    // Остальной код соответствует функции main из листинга 3-16
}
```

Мы проверяем присутствие отладчика в начале функции `main`. Если он обнаружен, процесс тестового приложения завершается вызовом `exit`. Такой способ использования `IsDebuggerPresent` неэффективен. Мы обнаружим отладчик только в том случае, если он запускает процесс приложения. Если же подключиться к уже запущенному процессу, мы сможем его отлаживать. В этом случае проверка `IsDebuggerPresent` уже произошла, а регулярного её повтора нет.

Листинг 3-18 демонстрирует правильный способ использования функции `IsDebuggerPresent`.

Листинг 3-18. Защита тестового приложения вызовом `IsDebuggerPresent`

```
#include <stdio.h>

int main()
{
    SHORT result = 0;

    while (gLife > 0)
    {
        if (IsDebuggerPresent())
        {
            printf("debugger detected!\n");
            exit(EXIT_FAILURE);
        }
        result = GetAsyncKeyState(0x31);
        if (result != 0xFFFF8001)
            --gLife;
        else
            ++gLife;

        printf("life = %u\n", gLife);
        Sleep(1000);
    }
    printf("stop\n");
    return 0;
}
```

Правильно вызывать `IsDebuggerPresent` на каждой итерации цикла `while` (например в его начале). Благодаря этому отладчик будет обнаружен, даже если он подключится к уже работающему приложению.

Как обойти такую защиту? Самый простой способ – манипулировать регистрами процессора в момент проверки. С помощью отладчика мы можем подменить возвращаемое функцией значение, чтобы предотвратить выполнение блока кода с вызовом `exit`.

Чтобы подменить результат вызова функции `IsDebuggerPresent`, выполните следующие действия:

1. Запустите отладчик OllyDbg и приложение из листинга 3-18 под его управлением.
2. Нажмите комбинацию клавиш Ctrl+N, чтобы открыть окно “Names in TestApplication” (имена в TestApplication). Перед вами **таблица символов** тестового приложения, в которой указаны все его глобальные переменные, константы и функции.
3. Введите имя `IsDebuggerPresent` в окне “Names in TestApplication”. При этом переход в списке к соответствующей функции произойдёт автоматически.
4. Щёлкните левой кнопкой мыши по строчке “&KERNEL32.IsDebuggerPresent” в списке.
5. Нажмите Ctrl+R, чтобы открыть диалог “Search - References to…” (поиск ссылок на…). Вы увидите список мест в коде приложения, из которых вызывается функция `IsDebuggerPresent`.
6. Двойным левым щелчком мыши выберите первую строчку в окне “Search - References to…”. Курсор окна дизассемблера перейдёт на вызов `IsDebuggerPresent` из функции `main`.
7. В окне дизассемблера левым щелчком мыши выберите инструкцию `TEST EAX,EAX`, которая следует за вызовом `IsDebuggerPresent`. Установите на ней точку останова нажатием F2.
8. Нажмите F9, чтобы продолжить работу тестового приложения. После этого должна сработать наша точка останова.
9. Измените значение регистра EAX на 0. Для этого двойным щелчком мыши выберите значение регистра EAX в окне “Registers (FPU)” (регистры). Откроется диалог “Modify EAX” (изменение EAX), как на иллюстрации 3-26. В нём введите значение 0 в ряд “Signed” (знаковый), столбец “EAX”. Нажмите кнопку “OK”.
10. Нажмите F9, чтобы приложение работало дальше.

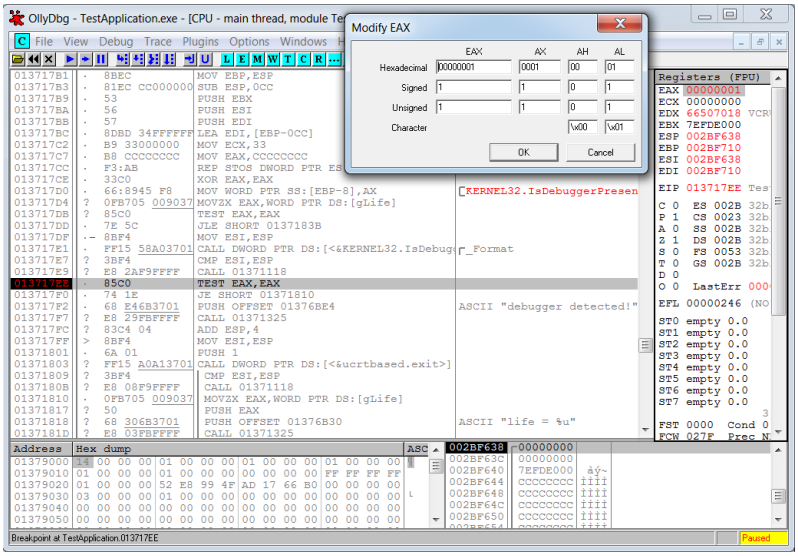


Иллюстрация 3-26. Изменение значения регистра EAX

После изменения значения регистра процессора, тестовое приложение не обнаружит отладчик на текущей итерации цикла `while`. Однако, проверка `IsDebuggerPresent` произойдёт на следующей итерации и OllyDbg будет обнаружен. Поэтому необходимо менять значение регистра вручную перед каждой проверкой, что неудобно.

Другой способ обойти проверку `IsDebuggerPresent` – модифицировать код тестового приложения. Сделать это можно как в исполняемом файле приложения на диске, так и в памяти уже работающего процесса. Второй способ удобнее в реализации, поэтому рассмотрим его. Как мы уже знаем, OllyDbg позволяет модифицировать память отлаживаемого процесса. Это может быть память любого сегмента: например данных в `.data` или кода в `.text`.

Чтобы модифицировать код приложения, выполните следующие действия:

1. Запустите отладчик OllyDbg и тестовое приложение из листинга 3-18 под его управлением.
2. Найдите место вызова функции `IsDebuggerPresent` в коде.
3. Выберите левым щелчком мыши инструкцию `JE SHORT 01371810`, следующую сразу за `TEST EAX,EAX` (см. иллюстрацию 3-27). Нажмите клавишу пробел, чтобы открыть диалог “Assemble” для её редактирования.
4. Измените инструкцию `JE SHORT 01371810` на `JNE SHORT 01371810` в диалоге, как показано на иллюстрации 3-27. После этого нажмите кнопку “Assemble”.
5. Нажмите F9, чтобы продолжить работу тестового приложения.

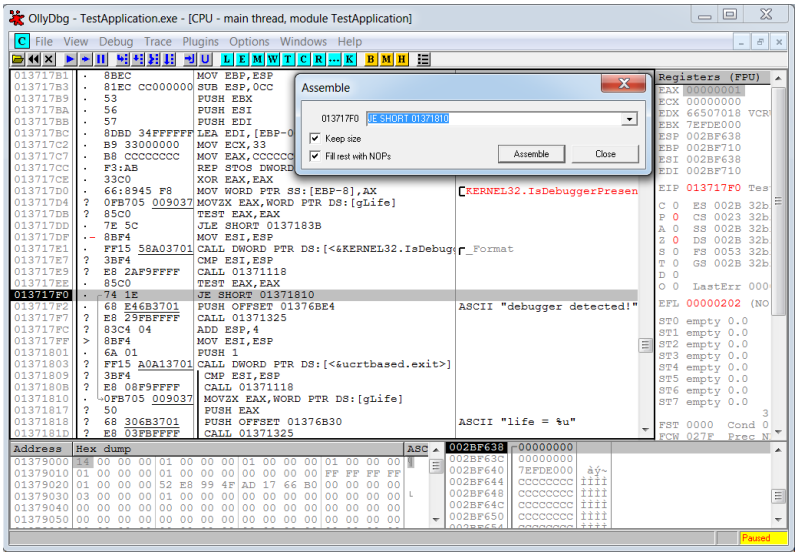


Иллюстрация 3-27. Диалог редактирования инструкции

После этих действий тестовое приложение больше не сможет обнаружить отладчик.

Что означает замена инструкции `JE` на `JNE`? Рассмотрим C++ код, соответствующий каждому варианту. Исходная инструкция `JE` аналогична следующему оператору `if`:

```
1 if (IsDebuggerPresent())
2 {
3     printf("debugger detected!\n");
4     exit(EXIT_FAILURE);
5 }
```

После замены инструкции на `JNE` мы получили такой код:

```
1 if ( ! IsDebuggerPresent())
2 {
3     printf("debugger detected!\n");
4     exit(EXIT_FAILURE);
5 }
```

Другими словами, мы инвертировали условие оператора `if`. Теперь если к тестовому приложению не подключён отладчик, оно завершится с сообщением “debugger detected!” (отладчик обнаружен) в консоль. Если же отладчик подключён, приложение продолжит свою работу.

После перезапуска тестового приложения, модификацию кода придётся повторить. Чтобы этого избежать, можно воспользоваться плагином **OllyDumpEx** отладчика OllyDbg. Он позволяет сохранить отредактированный код в исполняемый файл.

Для установки плагина OllyDumpEx выполните следующее:

1. Скачайте архив с плагином с сайта разработчика.

2. Распакуйте архив в папку установки OllyDbg. По умолчанию это:

```
C:\Program Files (x86)\odbg200
```

3. Проверьте путь до папки с плагинами в настройке OllyDbg. Для этого выберите пункт “Options” -> “Options…” главного меню. Откроется диалог “Options” (настройки). В левой его части выберите пункт “Directories” (каталоги). Поле “Plug-in directory” (каталог плагинов) должно соответствовать пути установки OllyDbg (например `C:\Program Files (x86)\odbg200`).
4. Перезапустите отладчик.

После этого в главном меню появится новый пункт “Plug-ins” (плагины). Чтобы воспользоваться возможностью сохранения модифицированного кода приложения в исполняемый файл, выполните следующее:

1. Выберите пункт главного меню “Plug-ins” -> “OllyDumpEx” -> “Dump process”. Откроется диалог “OllyDumpEx”.
2. В нём нажмите кнопку “Dump” (выгрузить). Откроется диалог “Save Dump to File” (сохранение дампа в память).
3. Укажите путь к исполняемому файлу для сохранения кода.

После этого на жёстком диске будет создан исполняемый файл с модифицированным кодом приложения. Его можно запустить как обычный EXE-файл. Он будет корректно работать в случае простого приложения. К сожалению, если это большая и сложная игра, она может завершиться с ошибкой после старта из дампа.

В интерфейсе WinAPI есть ещё одна функция для проверки подключённого отладчика – `CheckRemoteDebuggerPresent`. Она позволяет обнаружить отладчик, подключённый к указанному процессу. `CheckRemoteDebuggerPresent` может быть полезна, если система защиты и игра работают в разных процессах.

Обе функции `CheckRemoteDebuggerPresent` и `IsDebuggerPresent` проверяют данные PEB сегмента. `CheckRemoteDebuggerPresent` вызывает внутри себя WinAPI-функцию `NtQueryInformationProcess`, которая возвращает структуру типа `PROCESS_BASIC_INFORMATION`. Её второе поле – это указатель на структуру типа `PEB`. У `PEB` есть поле под названием `BeingDebugged`, значение которого равно 1, если к процессу подключён отладчик. Иначе значение поля равно 0.

CloseHandle

У функции `IsDebuggerPresent` есть два серьёзных недостатка. Во-первых, её вызовы легко обнаружить в исходном коде приложения и инвертировать условие проверки результата. Во-вторых, достаточно просто изменить значение поля `BeingDebugged` в PEB сегменте, чтобы предотвратить обнаружение отладчика.

Есть более изящные способы проверки наличия отладчика с помощью WinAPI. Один из них – использование побочного эффекта функции `CloseHandle`. Обычно `CloseHandle` вызывается, чтобы сообщить ОС об окончании работы с каким-то объектом. После этого объект может быть удалён, либо к нему могут получить доступ другие процессы. Очевидно, что любое сложное приложение интенсивно использует `CloseHandle`.

Функция `CloseHandle` имеет единственный входной параметр: дескриптор объекта. Если переданный дескриптор некорректен, будет сгенерировано **исключение** (exception) `EXCEPTION_INVALID_HANDLE`. То же самое произойдёт если процесс вызовет `CloseHandle` дважды для одного и того же дескриптора. Теперь важный момент – исключение генерируется только тогда, когда к процессу подключён отладчик. Если отладчика нет, исключения не будет и функция вернёт код ошибки. Таким образом мы можем следить за поведением функции и делать вывод о наличии отладчика.

Для обхода защиты, использующей `CloseHandle`, потребуется много работы. Прежде всего, надо отследить все вызовы функции. Затем надо отличить места, где с её помощью проверяется наличие отладчика. Во всех этих местах необходимо отредактировать код. Например, заменить вызов функции на `NOP` (no operation) инструкции процессора.

Пример использования `CloseHandle`:

```
1 BOOL IsDebug()
2 {
3     __try
4     {
5         CloseHandle((HANDLE)0x12345);
6     }
7     __except (GetExceptionCode() == EXCEPTION_INVALID_HANDLE ?
8               EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
9     {
10        return TRUE;
11    }
12    return FALSE;
13 }
```

Для обработки исключения `EXCEPTION_INVALID_HANDLE` мы применили конструкцию **try-except**, которая отличается от **try-catch**, определённой в стандарте языка C++. Эта конструкция – расширение для C и C++ от Microsoft, которое является частью механизма **Structured Exception Handling** (SEH).

Изменим наше тестовое приложение из листинга 3-18. Добавим определение функции `IsDebug` (приведённое выше) и будем вызывать её вместо `IsDebuggerPresent` в цикле `while`. Результат приведён в файле `CloseHandle.cpp` из примеров к книге. Попробуйте его скомпилировать и протестировать с отладчиками OllyDbg и WinDbg. Приложение успешно обнаруживает WinDbg, но не OllyDbg.

Это связано с тем, что OllyDbg имеет встроенный механизм для обхода такого типа защиты.

С помощью WinAPI-функции `DebugBreak` можно сделать очень похожую проверку на наличие отладчика:

```
BOOL IsDebug()
{
    __try
    {
        DebugBreak();
    }
    __except (GetExceptionCode() == EXCEPTION_BREAKPOINT ?
              EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
        return FALSE;
    }
    return TRUE;
}
```

В отличие от `CloseHandle`, `DebugBreak` всегда генерирует исключение `EXCEPTION_BREAKPOINT`. Если к приложению подключён отладчик, он обработает это исключение. Это значит, что блок `__except` приведённого выше кода не получит управление и функция `IsDebug` вернёт `TRUE`. Если же отладчика нет, исключение должно быть обработано приложением. В этом случае мы попадём в блок `__except` и функция вернёт значение `FALSE`.

Проверка на наличие отладчика через `DebugBreak` обнаруживает и OllyDbg, и WinDbg.

В WinAPI есть функция `DebugBreakProcess`, которая очень похожа на `DebugBreak`. Она позволяет сгенерировать исключение `EXCEPTION_BREAKPOINT` для указанного процесса. Это может быть полезно для реализации защиты, работающей в отдельном процессе.

CreateProcess

Есть метод запрещающий отладку процесса в принципе. Он связан со следующим ограничением ОС Windows: только один отладчик может быть подключён к процессу. Следовательно, если одна часть приложения подключается к другой в качестве отладчика, эта вторая часть становится защищённой. Этот метод известен как **самоотладка** (self-debugging).

Идея заключается в разделении приложения на два отдельных процесса: родительский и дочерний. При этом возможны следующие разделения обязанностей:

1. Дочерний процесс отлаживает родительский, который в свою очередь выполняет алгоритмы защищаемого приложения (TestApplication в нашем случае). Этот подход описан в статье.

2. Родительский процесс отлаживает дочерний. Дочерний выполняет алгоритмы защищаемого приложения.

Мы рассмотрим второй подход. Для создания дочернего процесса воспользуемся WinAPI-функцией `CreateProcess`. Полный код тестового приложения приведён в листинге 3-19.

Листинг 3-19. Защита тестового приложения методом самоотладки

```
#include <stdio.h>
#include <stdint.h>
#include <windows.h>
#include <string>

using namespace std;

static const uint16_t MAX_LIFE = 20;
static uint16_t gLife = MAX_LIFE;

void DebugSelf()
{
    wstring cmdChild(GetCommandLine());
    cmdChild.append(L" x");

    PROCESS_INFORMATION pi;
    STARTUPINFO si;
    ZeroMemory(&pi, sizeof(PROCESS_INFORMATION));
    ZeroMemory(&si, sizeof(STARTUPINFO));
    GetStartupInfo(&si);

    CreateProcess(NULL, (LPWSTR)cmdChild.c_str(), NULL, NULL, FALSE,
            DEBUG_PROCESS | CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);

    DEBUG_EVENT de;
    ZeroMemory(&de, sizeof(DEBUG_EVENT));

    for (;;)
    {
        if (!WaitForDebugEvent(&de, INFINITE))
            return;

        ContinueDebugEvent(de.dwProcessId,
                de.dwThreadId,
                DBG_CONTINUE);
    }
}

int main(int argc, char* argv[])
{
    if (argc == 1)
    {
        DebugSelf();
    }
    SHORT result = 0;

    while (gLife > 0)
    {
        result = GetAsyncKeyState(0x31);
        if (result != 0xFFFF8001)
            --gLife;
        else
            ++gLife;

        printf("life = %u\n", gLife);
        Sleep(1000);
    }

    printf("stop\n");
```

```
60      return 0;
61  }
```

Иллюстрация 3-28 демонстрирует взаимодействие родительского и дочернего процессов.

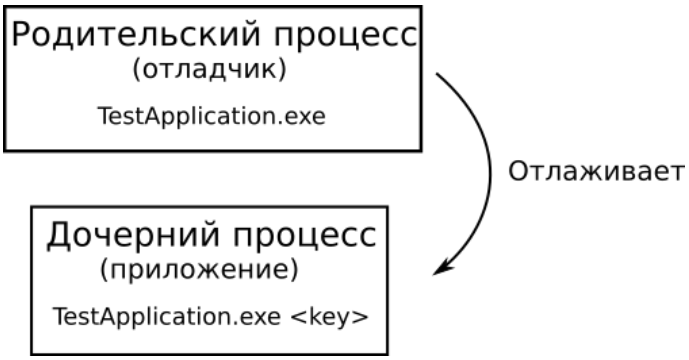


Иллюстрация 3-28. Взаимодействие родительского и дочернего процессов

Приложение из листинга 3-19 запускается в два этапа. Сначала пользователь щёлкает по иконке рабочего стола и приложение запускается без параметров командной строки. В этом случае следующее `if` условие будет истинным:

```
1      if (argc == 1)
2      {
3          DebugSelf();
4      }
```

Параметр `argv` функции `main` – это указатель на строку параметров командной строки. `argc` хранит их количество. Когда приложение запущено без параметров командной строки, `argc` равен 1, а строка `argv` содержит только имя запускаемого файла. Поэтому условие `if` истинно и приложение вызовет функцию `DebugSelf`. Это второй этап запуска приложения.

Алгоритм функции `DebugSelf` следующий:

1. Прочитать параметры командной строки и добавить к ним “x”. Этот параметр сообщает дочернему процессу, что он был запущен из родительского:

```
1 wstring cmdChild(GetCommandLine());
2 cmdChild.append(L" x");
```

2. Создать дочерний процесс с помощью вызова `CreateProcess`. В эту функцию мы передаём флаг `DEBUG_PROCESS`, который означает что новый процесс будет отлаживаться родительским. Также мы передаём флаг `CREATE_NEW_CONSOLE`, благодаря которому у дочернего процесса будет отдельная консоль. В ней вы сможете прочитать вывод нашего приложения.

3. Запустить бесконечный цикл `for`, в котором будем обрабатывать все события дочернего процесса.

Попробуйте запустить приложение из листинга 3-19 и подключиться к нему отладчиками OllyDbg и WinDbg. Ни одному из них это не удастся.

Наше тестовое приложение демонстрирует метод самоотладки в максимально простом и лаконичном виде. Его защиту очень просто обойти. Для этого достаточно запустить приложение из командной строки, передав параметром символ “x”:

```
1 TestApplication.exe x
```

В этом случае приложение запустится без самоотладки и к нему можно будет подключиться.

В настоящей защите нельзя полагаться на число параметров командной строки. Вместо этого следует проверять их значение. Например, родительский процесс может сгенерировать случайный ключ и передать его дочернему через вызов `CreateProcess`. Дочерний процесс проверяет корректность ключа при старте. В случае ошибки, работа приложения завершается.

Есть более надёжные техники обмена информацией между родительским и дочерним процессом, чем параметры командной строки. Они описаны в официальной документации Microsoft.

Операции с регистрами для обнаружения отладчиков

Все техники обнаружения отладчиков, использующие WinAPI-функции, имеют серьёзный недостаток: очень просто отследить места их вызовов. Даже если вы используете подход с `CloseHandle` и ваше приложение имеет тысячи вызовов этой функции, такую защиту можно обойти за предсказуемое время. Есть несколько техник, лишённых этого недостатка. Они основаны на манипуляции регистрами процессора. Доступ к этим регистрам можно получить через ассемблерные вставки или встроенные функции компилятора. Преимущество такого подхода в том, что анализ таблицы символов не поможет в поиске проверок на наличие отладчика. Из-за этого их намного сложнее обнаружить.

Флаг BeingDebugged

Рассмотрим, как функция `IsDebuggerPresent` устроена внутри. Мы знаем, что она проверяет данные PEB сегмента. Возможно, мы могли бы повторить её алгоритм.

Выполните следующие шаги для исследования функции `IsDebuggerPresent`:

1. Запустите отладчик OllyDbg.
2. Запустите из него тестовое приложение из листинга 3-18.

3. Найдите место вызова функции `IsDebuggerPresent` из `main`. Поставьте на нём точку останова. Продолжайте исполнение приложения.
4. Когда сработает точка останова нажмите F7, чтобы перейти к инструкциям функции `IsDebuggerPresent`.

В окне дизассемблера OllyDbg вы увидите код как на иллюстрации 3-29.

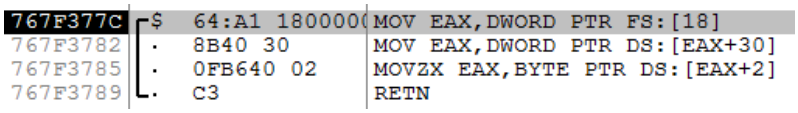


Иллюстрация 3-29. Инструкции функции `IsDebuggerPresent`

Рассмотрим каждую из четырёх инструкций функции `IsDebuggerPresent`:

1. Прочитать в регистр EAX базовый адрес TEB сегмента, соответствующего текущему потоку. Как мы уже знаем, регистр FS всегда указывает на сегмент TEB, а по смещению 0x18 в нём лежит собственный адрес.
2. Прочитать базовый адрес сегмента PEB в регистр EAX. Он хранится по смещению 0x30 в регистре TEB.
3. Прочитать значение флага `BeingDebugged` со смещением 0x2 из сегмента PEB в EAX регистр. По его значению можно определить наличие отладчика.
4. Вернуться из функции.

Повторим рассмотренный алгоритм в коде нашего тестового приложения. Результат приведён в листинге 3-20.

Листинг 3-20. Обнаружение отладчика через прямой доступ к PEB сегменту

```
#include <stdio.h>

int main()
{
    SHORT result = 0;

    while (gLife > 0)
    {
        int res = 0;
        __asm
        {
            mov eax, dword ptr fs:[18h]
            mov eax, dword ptr ds:[eax+30h]
            movzx eax, byte ptr ds:[eax+2h]
            mov res, eax
        };
        if (res)
        {
            printf("debugger detected!\n");
            exit(EXIT_FAILURE);
        }
        result = GetAsyncKeyState(0x31);
        if (result != 0xFFFF8001)
            --gLife;
        else
            ++gLife;

        printf("life = %u\n", gLife);
        Sleep(1000);
```

```
30      }
31      printf("stop\n");
32      return 0;
33  }
```

Сравните наш код и инструкции процессора на иллюстрации 3-29. Они почти одинаковы. Единственное отличие в последней инструкции. В нашем коде значение флага `BeingDebugged` присваивается переменной `res`. Сразу после ассемблерной вставки она проверяется в `if` условии.

Если вы поместите такую ассемблерную вставку и проверку на отладчик в нескольких местах приложения, их будет труднее найти чем вызовы функции `IsDebuggerPresent`. Можем ли мы в этом случае избежать дублирования кода? Это хороший вопрос. Если в следующих версиях Windows поменяется структура TEB или PEB сегмента, исправление придётся вносить в каждую копию ассемблерной вставки.

Есть несколько способов избежать дублирования кода. Очевидно, что в нашем случае мы не можем просто поместить его в обычную C++ функцию. Она обязательно попадёт в таблицу символов, по которой легко отследить все места её вызовов.

Можно вынести код ассемблерной вставки в C++ функцию и пометить её ключевым словом `__forceinline`. Такая функция называется **встроенной**. Компилятор будет вставлять её код в места вызовов. К сожалению, `__forceinline` игнорируется в нескольких случаях:

1. Приложение компилируется в конфигурации “Debug” (отладка).
2. Если встраиваемая функция содержит **рекурсивные вызовы**, т.е. вызывает саму себя.
3. Если встраиваемая функция делает вызов `alloca`.

Ключевое слово `__forceinline` работает только в конфигурации сборки “Release” (релиз), что может быть неудобно. В этом случае выходной исполняемый файл не содержит отладочной информации.

Альтернативное решение заключается в использовании макроса препроцессора. Компилятор вставляет тело макроса в каждое место исходного кода, где упоминается его имя. В этом случае поведение компилятора не зависит от конфигурации сборки.

Листинг 3-21 демонстрирует проверку флага `BeingDebugged` с помощью ассемблерной вставки, завёрнутой в макрос препроцессора.

Листинг 3-21. Обнаружение отладчика через прямой доступ к PEB сегменту

```
1 #include <stdio.h>
2
3 #define CheckDebug() \
```

```
4 int isDebugger = 0; \
5 { \
6 __asm mov eax, dword ptr fs : [18h] \
7 __asm mov eax, dword ptr ds : [eax + 30h] \
8 __asm movzx eax, byte ptr ds : [eax + 2h] \
9 __asm mov isDebugger, eax \
10 } \
11 if (isDebugger) \
12 { \
13 printf("debugger detected!\n"); \
14 exit(EXIT_FAILURE); \
15 }
16
17 int main()
18 {
19     SHORT result = 0;
20
21     while (gLife > 0)
22     {
23         CheckDebug();
24
25         result = GetAsyncKeyState(0x31);
26         if (result != 0xFFFF8001)
27             --gLife;
28         else
29             ++gLife;
30     }
31
32     printf("stop\n");
33
34     return 0;
35 }
```

Обратите внимание на использование макроса `CheckDebug` в функции `main`. Это выглядит как обычный вызов функции. Однако, поведения макроса и функции кардинально отличаются. Ещё на этапе обработки препроцессором файла с исходным кодом, который идёт до этапа компиляции, `main` будет преобразована следующим образом:

```
1 int main()
2 {
3     SHORT result = 0;
4
5     while (gLife > 0)
6     {
7         int res = 0;
8         __asm
9         {
10             mov eax, dword ptr fs:[18h]
11             mov eax, dword ptr ds:[eax + 30h]
12             movzx eax, byte ptr ds:[eax + 2h]
13             mov res, eax
14         };
15         if (res)
16         {
17             printf("debugger detected!\n");
18             exit(EXIT_FAILURE);
19         }
20         ...
```

Учитывайте эту особенность макросов при применении их в своих проектах. Если в теле макроса есть ошибка, компилятор укажет на строку его использования, а не определения.

Как вы помните, ассемблерные вставки не работают при компиляции 64-разрядных приложений на Visual Studio C++. В этом случае можно переписать макрос `CheckDebug` следующим образом:

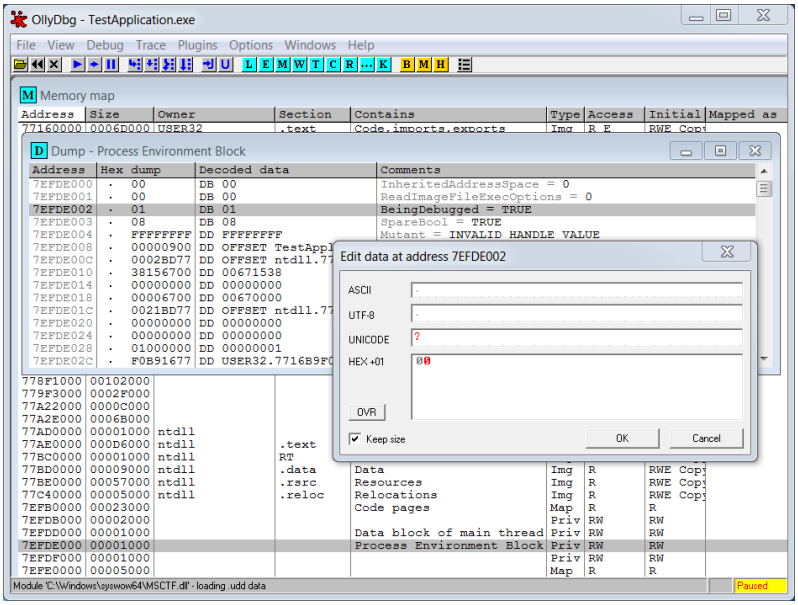
```
#include <winternl.h>

#define CheckDebug() \
{ \
PTEB pTeb = reinterpret_cast<PTEB>(__readgsqword(0x30)); \
PPEB pPeb = pTeb->ProcessEnvironmentBlock; \
if (pPeb->BeingDebugged) \
{ \
printf("debugger detected!\n"); \
exit(EXIT_FAILURE); \
} \
}
```

Не забудьте включить заголовочный файл `winternl.h`, в котором определены структуры `TEB` и `PEB`, а также указатели на них (`PTEB` и `PPEB`).

Защита, приведённая в листинге 3-21, выглядит достаточно надёжной. Так ли это и сможем ли мы её обойти? На самом деле это совсем несложно. Вместо того, чтобы искать в коде проверки и инвертировать `if` условия, мы можем просто изменить флаг `BeingDebugged` в PEB сегменте. Для этого выполните следующие шаги:

1. Запустите отладчик OllyDbg.
2. Из него запустите тестовое приложение из листинга 3-21.
3. Нажмите Alt+M, чтобы открыть карту памяти процесса. В ней найдите сегмент “Process Environment Block” (PEB).
4. Дважды щёлкните левой кнопкой мыши по сегменту PEB. Откроется окно “Dump - Process Environment Block”. В нём найдите значение флага “BeingDebugged”.
5. Щёлкните левой кнопкой мыши по флагу “BeingDebugged”, чтобы его выделить. Нажмите Ctrl+E – откроется диалог “Edit data at address…” (редактирование данных по адресу).
6. Измените значение поля “HEX+01” с “01” на “00” и нажмите кнопку “OK”, как изображено на иллюстрации 3-30.



Редактирование флага BeingDebugged

Если вы продолжите выполнение, приложение не обнаружит подключённый отладчик. Обход этой защиты очень прост. Поэтому рассмотрим более надёжный метод.

INT 3

Как вы помните, WinAPI-функция `DebugBreak` позволяет обнаружить отладчик по тому, кто обрабатывает сгенерированное ею исключение. Исследуем инструкции этой функции и попробуем повторить их с помощью ассемблерной вставки. Для этого выполните уже рассмотренные нами шаги, когда мы исследовали `IsDebuggerPresent`. Если вы сделаете всё правильно, то обнаружите, что функция `DebugBreak` состоит из единственной инструкции процессора `INT 3`. Именно она генерирует исключение `EXCEPTION_BREAKPOINT`.

Перепишем функцию `IsDebug` так, чтобы она использовала инструкцию `INT 3` вместо вызова `DebugBreak`:

```
BOOL IsDebug()
{
    __try
    {
        __asm int 3;
    }
    __except (GetExceptionCode() == EXCEPTION_BREAKPOINT ?
            EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
        return FALSE;
    }
    return TRUE;
}
```

Чтобы усложнить поиск вызовов функции `IsDebug`, мы могли бы применить ключевое слово `__forceinline` в её определении. Однако, в этом случае компилятор его проигнорирует. Дело в том, что обработчик `__try`/`__except` неявно выделяет блок памяти с помощью функции `alloca`. Как вы помните, это нарушает условие использования `__forceinline`.

Правильным решением будет использовать макрос:

```
#define CheckDebug() \
bool isDebugger = true; \
__try \
{ \
    __asm int 3 \
} \
__except (GetExceptionCode() == EXCEPTION_BREAKPOINT ? \
          EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) \
{ \
    isDebugger = false; \
} \
if (isDebugger) \
{ \
    printf("debugger detected!\n"); \
    exit(EXIT_FAILURE); \
}
```

Для 64-разрядного приложения воспользуемся встроенной функцией компилятора `__debugbreak()`:

```
#define CheckDebug() \
bool isDebugger = true; \
__try \
{ \
    __debugbreak(); \
} \
__except (GetExceptionCode() == EXCEPTION_BREAKPOINT ? \
          EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) \
{ \
    isDebugger = false; \
} \
if (isDebugger) \
{ \
    printf("debugger detected!\n"); \
    exit(EXIT_FAILURE); \
}
```

Вы можете найти файл с исходным кодом `Int3.cpp` тестового приложения, защищённого этим методом, в архиве примеров к книге. Чтобы обойти эту защиту, вам придётся найти все `if` проверки в коде и инвертировать их.

У OllyDbg есть функция поиска инструкций процессора в памяти отлаживаемого процесса. Для этого нажмите Ctrl+F в окне дизассемблера и в открывшемся диалоге введите значение “INT3”. После этого нажмите кнопку “Search” (поиск).

В машинном коде инструкция `INT 3` представляется шестнадцатеричным числом 0xCC. В результате поиска OllyDbg вы получите список инструкций, содержащих 0xCC в своём **коде операции** (opcode). Далеко не все из этих инструкций являются `INT 3`, но вам придётся их проверить.

Очевидно, рассмотренная нами защита не идеальна. Но для её преодоления придётся потратить много времени и усилий.

Проверка таймера

Во время отладки приложения, пользователь часто останавливает его выполнение. Обычно это нужно, чтобы проверить значения переменных, прочитать дамп памяти или дизассемблированный код. На этой особенности строится достаточно надёжная защита. Идея заключается в том, чтобы измерять время между контрольными точками в коде приложения. Если остановок выполнения не было, это время будет относительно небольшим (порядка миллисекунд). В противном случае можно с уверенностью утверждать, что приложение работает под отладчиком.

WinAPI-функции

Есть несколько WinAPI-функций, которые позволяют прочитать текущее время:

1. `GetTickCount` – возвращает количество миллисекунд с момента запуска ОС.
2. `GetLocalTime` – возвращает текущее время с учётом настройки часового пояса.
3. `GetSystemTime` – возвращает текущее всемирное координированное время (UTC).

Вы можете использовать любую из этих функций для замеров времени между контрольными точками. Листинг 3-22 демонстрирует решение с использованием `GetTickCount`.

Листинг 3-22. Замер времени между контрольными точками приложения с помощью `GetTickCount`

```
#include <stdio.h>
#include <stdint.h>
#include <windows.h>

static const DWORD MAX_DELTA = 1020;

static const uint16_t MAX_LIFE = 20;
static uint16_t gLife = MAX_LIFE;

int main()
{
    SHORT result = 0;

    DWORD prevCounter = GetTickCount();

    while (gLife > 0)
    {
        if (MAX_DELTA < (GetTickCount() - prevCounter))
        {
            printf("debugger detected!\n");
            exit(EXIT_FAILURE);
        }
        prevCounter = GetTickCount();

        result = GetAsyncKeyState(0x31);
        if (result != 0xFFFF8001)
            --gLife;
        else
            ++gLife;

        printf("life = %u\n", gLife);
        Sleep(1000);
    }
```

```
34
35      printf("stop\n");
36
37      return 0;
38  }
```

В этом примере мы измеряем время между итерациями цикла `while`. Если остановок не было, каждая итерация длится чуть больше одной секунды. Большую часть этого времени занимают вызовы `Sleep` (1000 миллисекунд) и `printf`. Если задержка оказывается больше константы `MAX_DELTA`, равной 1020 миллисекунд, скорее всего, была остановка. В этом случае приложение завершается.

Для тестирования примера выполните следующие действия:

1. Запустите отладчик OllyDbg.
2. Запустите из него приложение из листинга 3-22.
3. Начните выполнение процесса нажатием F9.
4. Остановите процесс нажатием F12.
5. Продолжите выполнение процесса по F9.

Приложение завершит свою работу с сообщением в консоль “debugger detected!” (отладчик обнаружен).

Чтобы обойти эту защиту, надо найти вызовы `GetTickCount` в коде приложения с помощью таблицы символов. Затем будет достаточно инвертировать проверку в операторе `if`.

Счётчики процессора

Текущее время можно читать не только с помощью WinAPI-функций. У процессора есть несколько аппаратных счётчиков. Один из них Time Stamp Counter (TSC), который считает количество тактовых сигналов (или циклов) с момента старта процессора. Его значение можно прочитать с помощью ассемблерных инструкций или встроенной функции компилятора.

Листинг 3-23 демонстрирует использование счётчика TSC для замеров времени между контрольными точками приложения.

Листинг 3-23. Замер времени между контрольными точками приложения с помощью TSC счётчика

```
1  #include <stdio.h>
2  #include <stdint.h>
3  #include <windows.h>
4
5  static const DWORD64 MAX_DELTA = 2650000000;
6
7  static const uint16_t MAX_LIFE = 20;
8  static uint16_t gLife = MAX_LIFE;
9
10 #define ReadRdtsc(result) \
11 { \
12 __asm cpuid \
13 __asm rdtsc \
14 __asm mov dword ptr[result + 0], eax \
15 __asm mov dword ptr[result + 4], edx \
```

```
}

int main()
{
    SHORT result = 0;

    DWORD64 prevCounter = 0;
    ReadRdtsc(prevCounter);

    while (gLife > 0)
    {
        DWORD64 counter = 0;
        ReadRdtsc(counter);

        if (MAX_DELTA < (counter - prevCounter))
        {
            printf("debugger detected!\n");
            exit(EXIT_FAILURE);
        }
        ReadRdtsc(prevCounter);

        result = GetAsyncKeyState(0x31);
        if (result != 0xFFFF8001)
            --gLife;
        else
            ++gLife;

        printf("life = %u\n", gLife);
        Sleep(1000);
    }

    printf("stop\n");

    return 0;
}
```

Для 64-разрядного приложения функция `main` будет выглядеть следующим образом:

TSC64.cpp

```
int main()
{
    SHORT result = 0;

    DWORD64 prevCounter = __rdtsc();

    while (gLife > 0)
    {
        DWORD64 counter = __rdtsc();

        if (MAX_DELTA < (counter - prevCounter))
        {
            printf("debugger detected!\n");
            exit(EXIT_FAILURE);
        }
        prevCounter = __rdtsc();

        result = GetAsyncKeyState(0x31);
        if (result != 0xFFFF8001)
            --gLife;
        else
            ++gLife;

        printf("life = %u\n", gLife);
        Sleep(1000);
    }

    printf("stop\n");

    return 0;
}
```

Алгоритм этой проверки точно такой же, как и в примере из листинга 3-22. Отличие только в способе замера времени и величине константы `MAX_DELTA`. В данном случае мы измеряем не миллисекунды, а тактовые сигналы процессора. Каждая итерация цикла длится примерно два с половиной миллиона циклов. Из-за этого пороговое значение `MAX_DELTA` получилось намного больше.

Обойти эту защиту труднее. Необходимо найти в коде приложения все инструкции `rdtsc` и выяснить, есть ли после каждой из них проверка на временную задержку. Если проверка есть, её надо инвертировать.

Защита приложения от ботов

В ОС Windows есть механизм **Security Descriptors** (SD) (дескрипторы безопасности) для ограничения доступа к системным объектам (например процессам). Он подробно описан в статье.

Следующие примеры демонстрируют использование SD:

- http://www.cplusplus.com/forum/windows/96406
- http://stackoverflow.com/questions/6185975/prevent-user-process-from-being-killed-with-end-process-from-process-explorer/10575889#10575889

В них приложение защищается с помощью **Discretionary Access Control List** (DACL) (дискреционный список контроля доступа). К сожалению, механизм SD не может защитить приложение, если к нему пытается получить доступ процесс, запущенный с правами администратора. В большинстве случаев пользователь, запускающий бота, имеет эти права. Поэтому мы не можем полагаться на ОС в вопросе защиты данных приложения и должны реализовывать собственные механизмы.

Надёжная система защиты должна решать две задачи:

1. Сокрытие данных от сканеров памяти (например Cheat Engine).
2. Проверка корректности данных для предотвращения их несанкционированного изменения.

Сокрытие данных

Рассмотрим техники сокрытия данных от сканеров памяти.

XOR шифр

Шифрование является одним из самых прямолинейных и надёжных способов защитить данные. Если состояния игровых объектов будут храниться в зашифрованном виде в памяти процесса, бот по-прежнему сможет их прочитать. Но это не значит, что он сможет восстановить актуальные параметры объектов.

XOR представляет собой самый простой алгоритм шифрования. Листинг 3-24 демонстрирует его использование.

Листинг 3-24. Защита данных приложения шифром XOR

```
#include <stdio.h>
#include <stdint.h>
#include <windows.h>

using namespace std;

inline uint16_t maskValue(uint16_t value)
{
    static const uint16_t MASK = 0xAAAA;
    return (value ^ MASK);
}

static const uint16_t MAX_LIFE = 20;
static uint16_t gLife = maskValue(MAX_LIFE);

int main(int argc, char* argv[])
{
    SHORT result = 0;

    while (maskValue(gLife) > 0)
    {
        result = GetAsyncKeyState(0x31);
        if (result != 0xFFFF8001)
            gLife = maskValue(maskValue(gLife) - 1);
        else
            gLife = maskValue(maskValue(gLife) + 1);

        printf("life = %u\n", maskValue(gLife));
        Sleep(1000);
    }

    printf("stop\n");

    return 0;
}
```

Функция `maskValue` шифрует данные при первом вызове и дешифрует при повторном. Чтобы получить зашифрованное значение, мы используем **операцию XOR** (также известную как “исключающее ИЛИ”) над данными и ключом. В качестве ключа используется константа `MASK`. Для расшифровки значения переменной `gLife`, `maskValue` вызывается повторно.

Если вы запустите приложение и попробуйте найти переменную `gLife` по её значению с помощью Cheat Engine, вам это не удастся. Однако, если значение константы `MASK` известно, задача значительно упрощается. Всё что вам нужно, это вручную или с помощью стандартного калькулятора Windows рассчитать зашифрованное значение `gLife` и задать его сканеру. В этом случае поиск даст результат.

Наша реализация шифра XOR упрощена в целях демонстрации подхода. Если вы планируете использовать её для защиты своих приложений, её следует доработать. Прежде всего будет полезно поместить алгоритм шифрования в шаблон класса (template) C++. Для этого класса следует определить арифметические операторы и

присваивание. Тогда вы сможете шифровать данные неявно и код будет выглядеть намного компактнее. Например так:

```
XORCipher<int> gLife(20);
gLife = gLife - 1;
```

Ещё одним улучшением будет генерация случайного ключа шифрования в конструкторе шаблона класса. Благодаря этому его будет труднее найти и применить для сканирования памяти.

Шифр AES

Даже с нашими улучшениями шифр XOR крайне прост для взлома. Чтобы надёжно защитить данные вашего приложения, понадобится более **криптостойкий** шифр. WinAPI предоставляет ряд криптографических функций. Среди них есть достаточно современный шифр AES. Попробуем применить его для нашего тестового приложения, как демонстрирует листинг 3-25.

Листинг 3-25. Защита данных приложения шифром AES

```
#include <stdint.h>
#include <stdio.h>
#include <windows.h>
#include <string>

#pragma comment (lib, "advapi32")
#pragma comment (lib, "user32")

using namespace std;

static const uint16_t MAX_LIFE = 20;
static uint16_t gLife = 0;

HCRYPTPROV hProv;
HCRYPTKEY hKey;
HCRYPTKEY hSessionKey;

#define kAesBytes128 16

typedef struct {
    BLOBHEADER  header;
    DWORD       key_length;
    BYTE        key_bytes[kAesBytes128];
} AesBlob128;

static const BYTE gCipherBlockSize = kAesBytes128 * 2;
static BYTE gCipherBlock[gCipherBlockSize] = {0};

void CreateContex()
{
    if (!CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_AES, CRYPT_VERIFYCONTEXT))
    {
        printf("CryptAcquireContext() failed - error = 0x%x\n", GetLastError());
    }
}

void CreateKey(string& key)
{
    AesBlob128 aes_blob;
    aes_blob.header.bType = PLAINTEXTKEYBLOB;
    aes_blob.header.bVersion = CUR_BLOB_VERSION;
    aes_blob.header.reserved = 0;
    aes_blob.header.aiKeyAlg = CALG_AES_128;
    aes_blob.key_length = kAesBytes128;
    memcpy(aes_blob.key_bytes, key.c_str(), kAesBytes128);
```

```
46
47      if (!CryptImportKey(hProv,
48                          reinterpret_cast<BYTE*>(&aes_blob),
49                          sizeof(AesBlob128),
50                          NULL,
51                          0,
52                          &hKey))
53      {
54          printf("CryptImportKey() failed - error = 0x%x\n", GetLastError());
55      }
56  }
57
58  void Encrypt()
59  {
60      unsigned long length = kAesBytes128;
61      memset(gCipherBlock, 0, gCipherBlockSize);
62      memcpy(gCipherBlock, &gLife, sizeof(gLife));
63
64      if (!CryptEncrypt(hKey, 0, TRUE, 0, gCipherBlock, &length, gCipherBlockSize))
65      {
66          printf("CryptEncrypt() failed - error = 0x%x\n", GetLastError());
67          return;
68      }
69      gLife = 0;
70  }
71
72  void Decrypt()
73  {
74      unsigned long length = gCipherBlockSize;
75
76      if (!CryptDecrypt(hKey, 0, TRUE, 0, gCipherBlock, &length))
77      {
78          printf("Error CryptDecrypt() failed - error = 0x%x\n", GetLastError());
79          return;
80      }
81      memcpy(&gLife, gCipherBlock, sizeof(gLife));
82      memset(gCipherBlock, 0, gCipherBlockSize);
83  }
84
85  int main(int argc, char* argv[])
86  {
87      CreateContex();
88
89      string key("The secret key");
90
91      CreateKey(key);
92
93      gLife = MAX_LIFE;
94
95      Encrypt();
96
97      SHORT result = 0;
98
99      while (true)
100     {
101         result = GetAsyncKeyState(0x31);
102
103         Decrypt();
104
105         if (result != 0xFFFF8001)
106             gLife = gLife - 1;
107         else
108             gLife = gLife + 1;
109
110         printf("life = %u\n", gLife);
111
112         if (gLife == 0)
113             break;
114
115         Encrypt();
116
117         Sleep(1000);
```

```
118     }
119     printf("stop\n");
120     return 0;
121 }
```

Рассмотрим алгоритм работы приложения. Его основные шаги вы можете проследить в функции `main`:

1. Создать контекст для криптографического алгоритма с помощью функции `CreateContex`. Это обёртка над WinAPI-функцией `CryptAcquireContext`. Контекст представляет собой комбинацию двух компонентов: **контейнер ключей** и **Cryptography Service Provider** (CSP) (криптопровайдер). Контейнер содержит все ключи, принадлежащие пользователю. CSP – это программный модуль, реализующий криптографический алгоритм.
2. Добавить ключ шифрования в CSP с помощью функции `CreateKey`. Функция принимает в качестве входного параметра строку со значением ключа. Из неё создаётся структура BLOB (расшифровывается как Binary Large Object, т.е. двоичный большой объект). Эта структура передаётся в CSP с помощью WinAPI вызова `CryptImportKey`.
3. Инициализировать переменную `gLife` и зашифровать её функцией `Encrypt`. Внутри себя она вызывает WinAPI-функцию `CryptEncrypt`. Зашифрованное значение сохраняется в глобальном байтовом массиве `gCipherBlock`. При этом значение переменной `gLife` зануляем, чтобы сканер памяти не смог её найти.
4. Перед каждым использованием переменной `gLife` расшифровываем её значение функцией `Decrypt`, которая вызывает внутри себя WinAPI-функцию `CryptDecrypt`. После работы с `gLife` мы снова её шифруем.

В чём преимущество шифра AES по сравнению с XOR? На самом деле алгоритм поиска зашифрованного значения в памяти одинаков в обоих случаях:

1. Восстановить ключ шифрования.
2. Применить ключ для шифровки текущего значения переменной.
3. Искать зашифрованное значение в памяти процесса с помощью сканера.

XOR шифр работает намного быстрее, но его проще взломать. Для этого есть два варианта: перебор всех возможных ключей или поиск ключа в памяти процесса. В некоторых случаях первый подход будет быстрее и проще. Для шифра AES есть только один вариант – поиск ключа в памяти. Чтобы взломать его перебором, понадобится значительное время. Поэтому стойкость защиты определяется только тем, насколько хорошо спрятан ключ. Надёжным решением может быть генерация нового ключа при каждом запуске приложения.

У шифра AES есть ещё одно достоинство. После восстановления ключа, необходимо точно повторить алгоритм шифрования. Только так возможно получить зашифрованное значение из того, которое отображается в окне игры. Шифр XOR настолько прост, что вы можете вычислить зашифрованное значение в уме. AES же

использует несколько этапов применения операций XOR и битового сдвига. Потребуется специальное приложение для выполнения шифрования, а для его разработки нужны время и знания.

Оба шифра XOR и AES скрывают данные приложения от сканирования. Это значит, что боту будет сложно найти информацию об объектах в памяти процесса. Однако, это не помешает ему писать произвольные данные в память. В некоторых случаях это может стать уязвимостью.

Проверка корректности данных

Теперь рассмотрим способы защиты данных приложения от несанкционированного изменения. Идея заключается в том, чтобы дублировать данные и периодически сравнивать их с копией. Наше приложение должно модифицировать данные и копию одновременно. Если в какой-то момент времени они различаются, можно заключить, что изменение было сделано не приложением, а сторонней программой.

Если значения данных и копии всегда одинаковы, копию будет легко найти в памяти процесса с помощью сканера. Тогда бот будет знать её месторасположение и менять вместе с данными. Таким образом, наша задача заключается в том, чтобы скрыть копию. Для этой цели мы могли бы применить шифрование, но есть более быстрый способ трансформации данных – **хеширование** (hashing).

Хеширование очень похоже на шифрование. Алгоритм берёт исходные данные и конвертирует их в другое представление. Различие заключается в том, что шифрование обратимо, т.е. данные можно расшифровать и получить исходное значение. Операция же хеширования необратима. Благодаря этому свойству алгоритмы хеширования работают намного быстрее.

Проверка целостности данных с помощью хеширования приведена в листинге 3-26.

Листинг 3-26. Проверка целостности данных приложения

```
#include <stdio.h>
#include <stdint.h>
#include <windows.h>
#include <functional>

using namespace std;

static const uint16_t MAX_LIFE = 20;
static uint16_t gLife = MAX_LIFE;

std::hash<uint16_t> hashFunc;
static size_t gLifeHash = hashFunc(gLife);

void UpdateHash()
{
    gLifeHash = hashFunc(gLife);
}

__forceinline void CheckHash()
{
    if (gLifeHash != hashFunc(gLife))
    {
        printf("unauthorized modification detected!\n");
```

```
24          exit(EXIT_FAILURE);
25      }
26  }
27
28  int main(int argc, char* argv[])
29  {
30      SHORT result = 0;
31
32      while (gLife > 0)
33      {
34          result = GetAsyncKeyState(0x31);
35
36          CheckHash();
37
38          if (result != 0xFFFF8001)
39              --gLife;
40          else
41              ++gLife;
42
43          UpdateHash();
44
45          printf("life = %u\n", gLife);
46
47          Sleep(1000);
48      }
49
50      printf("stop\n");
51
52      return 0;
53  }
```

В этом примере мы добавили вспомогательную переменную `gLifeHash`, которая хранит хэшированное значение `gLife`. Для вычисления хеша используется функция `hash` из **стандартной библиотеки шаблонов** (STL) стандарта C++11.

На каждой итерации `while` цикла мы сравниваем хэшированное и текущее значение переменной `gLife` в функции `CheckHash`. Если они различаются, мы делаем вывод о несанкционированном изменении переменной. После проверки мы работаем с `gLife` точно так же, как и раньше. Затем пересчитываем её хеш с помощью функции `UpdateHash` и назначаем новое значение `gLifeHash`.

Попробуйте скомпилировать и запустить этот пример. Если вы модифицируете значение переменной `gLife` с помощью сканера Cheat Engine, приложение завершит свою работу.

Обойти такую защиту возможно. Для этого бот должен одновременно модифицировать переменные `gLife` и `gLifeHash`. Но здесь есть подводные камни. Во-первых, хэшированное значение не так-то просто обнаружить. Если алгоритм известен, вы можете рассчитать хеш исходного значения и найти его с помощью сканера памяти. В большинстве случаев алгоритм неизвестен. Чтобы его восстановить надо проанализировать дизассемблированный код приложения. Во-вторых, необходимо выбрать правильный момент для модификации. Если запись нового значения происходит во время проверки `if` в функции `CheckHash`, изменение будет обнаружено.

Вместо того, чтобы искать хэшированное значение и модифицировать его одновременно с исходным, можно инвертировать все проверки `if` алгоритма защиты. Но если функция наподобие `CheckHash` будет встроенной или заменена макросом, то найти эти проверки будет трудно.

Надёжнее всего данные игры будут защищены от несанкционированного изменения, если они хранятся на стороне сервера. В этом случае клиент получит их только для визуализации текущего состояния игры. Изменение этих данных ботом повлияет на картинку в окне приложения, но их копия на стороне сервера останется неизменной. Можно ожидать, что эта копия всегда будет корректна. Если данные клиента в какой-то момент будут отличаться, их можно восстановить из копии.

Выводы

Мы рассмотрели методы защиты памяти процесса игрового приложения. Большинство из них можно реализовать с помощью WinAPI-функций. Однако, в некоторых случаях операции с регистрами позволят лучше скрыть алгоритм защиты от исследования.

Мы познакомились с методами защиты от отладки и сканирования памяти, а также с техниками предотвращения несанкционированного изменения данных приложения.

Внеигровые боты

В этой главе мы познакомимся с внеигровыми ботами. Сначала рассмотрим инструменты для их разработки. После этого изучим основные принципы работы вычислительных сетей. Попробуем написать простое сетевое приложение. Когда мы освоим инструменты разработки, напишем внеигрового бота для существующей игры. В конце главы рассмотрим методы защиты от ботов этого типа.

Инструменты для разработки

Инструменты для разработки внутриигровых и внеигровых ботов различаются. В первом случае нам нужны эффективные средства для доступа к памяти процесса игры и манипуляции его данными. Внеигровые боты полностью замещают собой игровой клиент и дублируют его основные возможности.

Язык программирования

Многие из существующих внеигровых ботов написаны на C++. Этот язык хорошо интегрируется с WinAPI, а кроме того для него существует много сторонних библиотек, в том числе для работы с сетью и криптографией. C++ – отличный инструмент для разработки ботов. Но в этой главе мы воспользуемся другим языком для наших примеров.

Мы будем использовать скриптовый язык **Python** по нескольким причинам. Прежде всего, он лаконичнее C++. Благодаря этому наши примеры станут короче и понятнее для чтения. Также у Python есть библиотеки (известные как **модули**) для работы с сетью и криптографией. Эти возможности очень важны для разработки внеигровых ботов.

Для работы с Python подойдёт практически любая IDE. Я предпочитаю Notepad++, которым мы пользовались во второй главе.

Есть два варианта установки Python и криптографической библиотеки. Первый вариант – Python последней версии 3.6.5 и библиотека PyCryptodome. PyCryptodome – это **ответвлённый проект** библиотеки

PyCrypto. В нём лучше реализована поддержка ОС Windows. К сожалению, этот проект не имеет некоторых устаревших возможностей PyCrypto. Они вряд ли понадобятся при разработке реальных ботов, но могут быть полезны для учебных целей при знакомстве с криптографией. Второй вариант установки подразумевает более старую версию Python 3.3.0 и библиотеку PyCrypto.

Все примеры этой главы корректно исполняются на обеих версиях Python 3.6.5 и 3.3.0. Но если вы выберите вариант с PyCryptodome, вы не сможете запустить несколько примеров. Они не так важны, и будет достаточно просто рассмотреть их код.

Для установки Python 3.3.0 и библиотеки PyCrypto выполните следующие действия:

1. Скачайте Python 3.3.0 с официального сайта.
2. Установите Python. Выберите путь установки по умолчанию: `C:\Python33`.
3. Скачайте неофициальную сборку библиотеки PyCrypto.
4. Установите библиотеку. В процессе установки Python будет найден автоматически.

Инструкция по установке Python 3.6.5 и библиотеки PyCryptodome:

1. Скачайте Python 3.6.5 с официального сайта.
2. Установите его по пути по умолчанию: `C:\Program Files\Python36`.
3. Скачайте скрипт `get-pip.py` с сервера bootstrap. Этот скрипт устанавливает менеджер модулей `pip`. С его помощью вы сможете скачивать нужные вам модули Python.
4. Запустите `get-pip.py` из командной строки:

```
1 get-pip.py --user
```

Когда скрипт закончит свою работу, вы увидите сообщение с путём установки менеджера pip. В моём случае это `C:\Users\ilya.shpigor\AppData\Roaming\Python\Python36\Scripts`.

5. Перейдите по пути установки pip и запустите его:

```
1 pip install --user pycryptodome
```

По этой команде будет скачана библиотека PyCryptodome.

После установки любой версии Python нужно проверить, что путь до интерпретатора `python.exe` попал в переменную окружения `PATH`. Для этого выполните следующие действия:

1. Откройте диалог “Control Panel” -> “System” -> “Advanced system settings” (“Панель управления” -> “Система” -> “Дополнительные параметры системы”). Нажмите кнопку “Environment Variables” (переменные среды). Вы увидите диалог с двумя списками.
2. В списке “System variables” (переменные системы) найдите переменную “PATH”. Выберите её левым щелчком мыши.
3. Нажмите кнопку “Edit” (Редактирование). Вы увидите текущий список путей в переменной `PATH`.
4. Добавьте в список ваш путь установки Python, если его там нет.

Теперь ваша система готова к запуску примеров этой главы.

Язык Python кросс-платформенный. Это значит, что написанные на нём скрипты можно запускать на Windows, Linux и macOS с незначительными изменениями.

Анализатор трафика

Wireshark – один из самых известных анализаторов трафика с открытым исходным кодом. Благодаря ему вы сможете перехватывать весь входящий и исходящий трафик с указанной сетевой платы, просматривать его в удобном интерфейсе пользователя, фильтровать пакеты, выводить статистику и сохранять результат на жёстком диске. Кроме этого, Wireshark имеет функции для интерпретации данных и расшифровки большинства сетевых протоколов.

Конфигурация Windows

В этой главе мы будем работать с сетевыми приложениями. Каждое из них состоит из двух частей (клиент и сервер), запущенных на разных компьютерах, которые соединены друг с другом через сеть. Для тестирования таких приложений нужны либо два компьютера, либо специальные средства вроде виртуальной машины. В этом случае одна часть приложения запускается на хост-системе (ваша ОС), а другая часть в

виртуальной машине (гостевая система). Системы подключаются друг к другу через эмулируемую локальную сеть.

К счастью, у современных ОС есть возможность запуска и отладки сетевых приложений без вспомогательных компьютеров или виртуальных машин. Для этой цели служит специальный сетевой интерфейс, известный как **loopback** (петля). Обе части сетевого приложения, запущенные на одном компьютере могут обмениваться сетевыми пакетами через loopback. При этом они ведут себя практически так же, как если бы взаимодействовали через реальную сеть.

По умолчанию интерфейс loopback отключён в Windows. Чтобы запустить наши тестовые примеры, вам потребуется его включить. Для этого выполните следующие шаги:

1. Запустите Device Manager (диспетчер устройств). Вы можете сделать это через Control Panel (панель управления) или набрав команду “Device Manager” в меню Start (пуск).
2. Выберите корневой элемент в дереве устройств окна Device Manager.
3. Выберите пункт меню “Action” -> “Add legacy hardware” (“Действие” -> “Установить старое устройство”). Откроется диалог “Add Hardware” (установить устройство).
4. Нажмите кнопку “Next” (далее) на первой странице диалога.
5. На второй странице диалога выберите пункт “Install the hardware that I manually select from a list (Advanced)” (установка оборудования, выбранного из списка вручную). Нажмите кнопку “Next”.
6. В списке “Common hardware types” (стандартные типы оборудования) выберите пункт “Network adapters” (сетевые платы). Нажмите кнопку “Next”.
7. Выберите производитель “Microsoft” и сетевую плату “Microsoft Loopback Adapter”. Нажмите кнопку “Next” на этой и следующей страницах.
8. Когда процесс установки завершится, нажмите кнопку “Finish” (завершить).

После установки интерфейса loopback, его необходимо включить. Для этого выполните следующие действия:

1. Откройте окно “Network and Sharing Center” (центр управления сетями и общим доступом). Это можно сделать через меню “Start”.

2. Щёлкните по пункту “Change adapter settings” (изменение параметров адаптера) в левой части окна. Откроется новое окно “Network Connections” (сетевые подключения).
3. Правым щелчком мыши по иконке “Microsoft Loopback Adapter” откройте всплывающее меню. В нём выберите пункт “Enable” (включить).

Теперь интерфейс loopback готов к работе.

Сетевые протоколы

В первой главе мы рассмотрели архитектуру типичной онлайн-игры. Как вы помните, в ней игровой клиент взаимодействует с сервером через сеть (в большинстве случаев это Интернет). Для передачи пакетов клиент вызывает функции WinAPI. ОС обрабатывает эти вызовы и отправляет указанные данные по сети. На аппаратном уровне для этого используется сетевая плата, функции которой доступны ОС благодаря драйверу устройства.

Возникает вопрос: как именно происходит передача данных по сети? Попробуем найти на него ответ вместе.

Задачи при передаче данных

Чтобы лучше понять существующие решения в какой-то технической области, будет разумным рассмотреть решаемые ими задачи. Представим, что мы с вами разработчики программ и нам поставили задачу передать данные игрового клиента на сервер через существующую сеть.

У нас есть два устройства, подключённых к сети как на иллюстрации 4-1. Они называются **сетевыми хостами**.

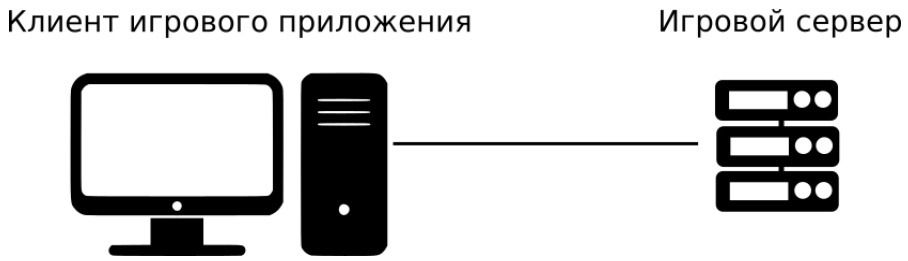


Иллюстрация 4-1. Игровой клиент и сервер, соединённые сетью

Самое прямолинейное и простое решение – реализовать алгоритм передачи данных целиком в игровом клиенте. Этот алгоритм может выглядеть следующим образом:

1. Скопировать все состояния игровых объектов в байтовый массив. Такой массив называется **сетевым пакетом**.
2. Скопировать подготовленный пакет в память, доступную для сетевой платы. Обычно эта память работает в режиме DMA.
3. Дать плате команду на отправку пакета.

Наш алгоритм успешно справляется с передачей данных до тех пор, пока сеть состоит только из двух устройств. Но что произойдёт, если подключить третий хост как на иллюстрации 4-2?

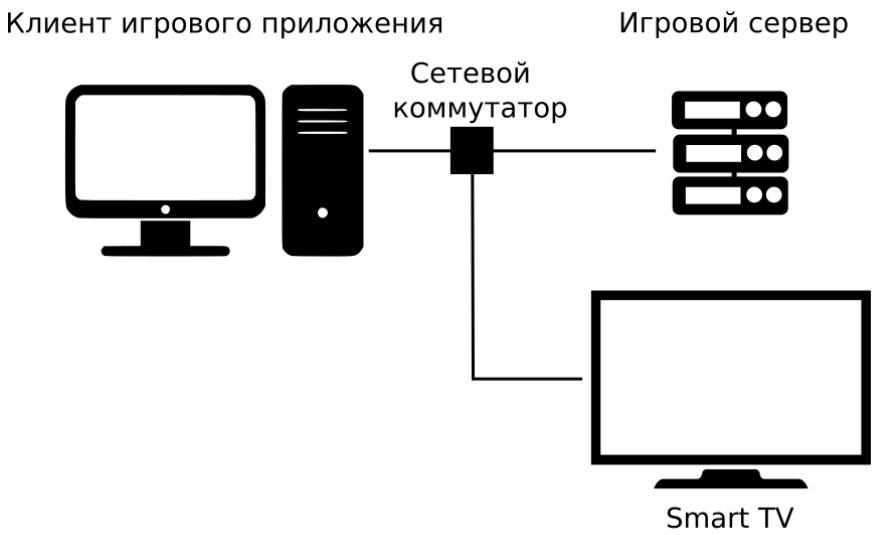


Иллюстрация 4-2. Сеть из трёх хостов

В этому случае нам не обойтись без дополнительного устройства, известного как **сетевой коммутатор** (network switch). У обычной современной сетевой платы Ethernet есть только один порт. Она рассчитана на подключение точка-точка. Поэтому трёх сетевых плат просто не хватит для сети из трёх хостов. Конечно, можно установить несколько сетевых плат на каждый компьютер, но это будет слишком дорого. Сетевой коммутатор решает проблему. На данный момент будем рассматривать его, только как средство физического подключения нескольких хостов к одной сети.

После появления третьего устройства в сети возникла проблема. Каким-то образом необходимо различать хосты и направлять игровые данные от клиента на сервер, а не на телевизор. Вы можете возразить, что нет ничего плохого, если телевизор получит несколько ненужных ему пакетов. Он может их просто проигнорировать. Эта мысль верна до тех пор, пока наша сеть небольшая. Но что случится, если к ней подключатся сотни хостов? Если каждый узел будет посылать трафик для каждого, сеть окажется перегружена. Задержки в передаче пакетов станут настолько велики, что никакого эффективного взаимодействия между хостами не получится. Причина этого в том, что сетевые кабели и платы имеют ограниченную пропускную способность в силу аппаратных особенностей. С этим ресурсом нам следует работать осмотрительно.

Проблему различия хостов в сети можно решить, если каждому из них назначить уникальный идентификатор. Мы пришли к первому решению, которое приняли настоящие разработчики сетей. **MAC-адрес** – это уникальный идентификатор сетевой платы или другого передающего в сеть устройства. Этот адрес неизменный и назначается изготовителем на этапе производства устройства. Теперь наше игровое приложение может добавлять MAC-адрес целевого хоста к каждому передаваемому пакету. Благодаря этому сетевой коммутатор сможет перенаправлять пакет только на тот свой порт, к которому подключён целевой хост.

Откуда коммутатор знает MAC-адреса хостов подключённые к его портам? Для этого он следит за всеми входящими на каждый порт пакетами. Из них он читает MAC-адрес отправителя и добавляет его в таблицу разрешения адресов, также известную как Address Resolution Logic (ARL). В этой таблице каждая строка содержит MAC-адрес и соответствующий ему порт.

Когда сервер получит пакет клиента, он захочет подтвердить корректность принятых данных, либо в случае ошибки запросить повторной передачи. Для этого нужно знать MAC-адрес отправителя. Поэтому будет разумным при отправке пакета клиентом добавлять не только MAC-адрес целевого хоста, но и свой собственный.

Предположим, что наша сеть стала больше. Например, к ней подключены хосты, находящиеся в двух расположенных недалеко друг от друга зданиях. Каждое из них имеет собственную локальную сеть (или подсеть),

состоящую для простоты из трёх компьютеров. Обе они объединены в единую сеть через **маршрутизатор** (router), как на иллюстрации 4-3.

Иллюстрация 4-3. Две локальные сети, соединённые маршрутизатором

На самом деле в каждой из двух локальных сетей могут быть десятки хостов. Если мы по-прежнему будем использовать MAC-адреса для указания целей пакетов, возникнут сложности. Каждый хост должен знать адреса всех получателей, с которыми он обменивается данными. Самое простое решение этой проблемы заключается в том, чтобы хранить список MAC-адресов всех хостов в сети на каждом из них. Тогда при подключении нового компьютера надо выполнить следующие действия:

1. Добавить MAC-адрес нового хоста во все существующие списки.
2. Скопировать исправленный список на новый хост.

Не забывайте также об исправлении списков адресов, когда один из хостов отключается. Очевидно, что вручную поддерживать эти списки в актуальном состоянии очень трудоёмко.

Вместо ручной правки и копирования списков можно написать алгоритм автоматического обнаружения хостов. Например, только что подключившийся к сети компьютер отправляет **широковещательный запрос** всем остальным. Любой, кто получает этот запрос, должен выслать свой MAC-адрес отправителю. Подобный механизм существует и известен как **протокол определения адреса** (Address Resolution Protocol или ARP). На самом деле ARP работает несколько сложнее. Когда какой-то хост хочет начать обмен данными, но не знает MAC-адрес получателя, он отправляет широковещательный запрос. В этом запросе указано (по IP-адресу о котором далее), кто именно должен на него ответить. Таким образом отвечает только тот хост, которого ищут.

Что означает термин “протокол” применительно к сетям? Это набор соглашений о формате данных. Например, наше приложение посылает игровые данные на сервер. Должны ли мы добавлять MAC-адреса отправителя и получателя в начале сетевого пакета или в конце? Если в начале – получатель должен знать об этом решении и интерпретировать первые байты пакета как адреса. Кроме того протокол определяет, как будут обрабатывать ошибки передачи данных. Например, сервер получает только половину отправленного клиентом пакета. Логично будет запросить его повторную передачу. Чтобы это сработало, клиент должен правильно понять сообщение от сервера о потере пакета. Спецификация протокола включает в себя все подобные нюансы взаимодействия сетевых хостов.

Вернёмся к нашей разросшейся сети. Очевидно, мы имеем некоторое дублирование данных, поскольку все хосты знают друг друга и должны хранить таблицу MAC-адресов в своей памяти. Протокол ARP помогает частично решить эту проблему. Благодаря ему актуальность таблиц будет поддерживаться динамически. Но их размер станет значительным, если сеть насчитывает десятки тысяч хостов. Было бы намного эффективнее, если бы только хосты одной подсети знали друг друга. При обмене данными между компьютерами из разных подсетей, маршрутизатор мог бы перенаправлять их пакеты. Таким образом хостам нужно будет знать только свою подсеть, частью которой является маршрутизатор.

Чтобы решить проблему с дублированием данных в таблицах, нам нужно что-то более гибкое чем MAC-адреса. Для передачи пакетов между подсетями был бы удобен механизм назначения хостам произвольных идентификаторов. Тогда мы могли бы назначить определённый диапазон “адресов” компьютерам одной подсети. Зная правило выбора диапазона, маршрутизатор мог бы быстро вычислять подсеть получателя по идентификатору и перенаправлять пакет. Мы говорим об уже существующем решении, известном как **IP-адреса**.

Теперь наше игровое приложение и сервер могут эффективно взаимодействовать, даже находясь в разных подсетях. Но что случится если мы запустим чат-программу на том же компьютере, где уже работает игровой клиент? Оба приложения должны посылать и принимать сетевые пакеты. Когда ОС получает пакет, указанные в нём IP- и MAC-адреса соответствуют текущему хосту. Однако, этой информации недостаточно,

чтобы найти программу-получатель среди работающих в данный момент. Для решения этой проблемы нужно добавить некий идентификатор приложения. Он называется **портом**. В каждом сетевом пакете должны быть указаны порты приложения отправителя и получателя. Тогда ОС сможет гарантировать правильность передачи пакета ожидающему его процессу. Порт отправителя нужен, чтобы получатель смог ответить.

Возможно, вы уже заметили, что реализация нашего игрового приложения становится слишком сложной. Оно должно подготовить пакет, содержащий состояния игровых объектов, MAC-адреса, IP-адреса и порты. Также было бы полезно подсчитать **контрольную сумму** передаваемых данных и поместить её в тот же пакет. Приложение на стороне сервера должно иметь те же самые алгоритмы для кодирования и декодирования адресов, портов, игровых данных, а также подсчёта контрольной суммы. Эти алгоритмы выглядят достаточно универсальными. Любое приложение (например чат-программа или браузер) могло бы использовать их для передачи своих данных. В то же время каждый хост сети должен иметь эти алгоритмы. Лучшим решением будет поместить их в библиотеки ОС.

Мы пришли к решению, известному как **стек протоколов**. Этот термин означает реализацию набора сетевых протоколов. Слово “стек” используется, чтобы подчеркнуть иерархическую зависимость одних протоколов от других. Каждый из них относится к одному из **уровней** иерархии. При этом низкоуровневые протоколы предоставляют свои возможности для высокоуровневых. Например, стандарт IEEE 802.3 описывает правила передачи данных на физическом уровне по витой паре, а стандарт IEEE 802.11 - для беспроводной связи Wi-Fi. Протоколы уровней выше должны уметь передавать данные по обоим типам соединений. Это означает, что на каждом уровне может быть реализовано несколько взаимозаменяемых протоколов. В зависимости от требований пользователь может выбрать протокол подходящий для его задачи. Когда возникает разнообразие реализаций, крайне важно чётко определить обязанности каждого уровня. Именно для этого была создана **сетевая модель OSI** (Open Systems Interconnection).

Мы кратко рассмотрели основные решения современных сетевых коммуникаций. Теперь у нас достаточно знаний, чтобы изучить реальный стек протоколов, используемый сегодня в сети Интернет.

Стек протоколов TCP/IP

Почему мы собираемся рассмотреть **стек TCP/IP**, когда речь зашла об Интернете? Возможно, вы ожидали, что в самой большой сети на планете должен использоваться стек, строго построенный по OSI модели. Ведь на её создание у двух интернациональных комитетов (ISO и CCITT) ушло несколько лет. В результате они разработали хорошо продуманный стандарт, покрывающий все возможные требования по взаимодействию в сети.

Было несколько попыток применить модель OSI на практике и реализовать протоколы для каждого её уровня. Все эти проекты не увенчались успехом. Главная проблема заключается в том, что модель OSI избыточна. Многие её функции оказались не нужны при практическом применении. В результате сетевые пакеты содержали никем не используемые данные, а это лишние накладные расходы.

Ещё одна проблема модели заключается в частичном перекрытии обязанностей некоторых уровней. Как результат в сетевом пакете оказываются дублирующиеся данные, используемые разными протоколами. Алгоритмы для их обработки копируются, что приводит к увеличению объёма исполняемого кода. Это также негативно отражается на быстродействии. Разработчикам требуется больше усилий на написание и сопровождение стека протоколов. Всё это приводит к его удорожанию.

Пока велась работа над моделью OSI, два исследователя Роберт Кан и Винтон Серф создали стек протоколов TCP/IP. Это произошло на несколько лет раньше публикации стандарта OSI. Роберт и Винтон занимались конкретной практической задачей – передачей данных в сети ARPANET. Возможно, благодаря этому их решение оказалось эффективным и простым в реализации. Впоследствии этот стек был опубликован комитетом IEEE в качестве открытого стандарта, получившего название **модель TCP/IP**, в 1974 году. Модель OSI увидела свет только в 1984.

Сразу после публикации модели TCP/IP разработчики энтузиасты и компании начали реализовывать собственные версии стека для существовавших в то время ОС. Он оказался настолько прост, что программист в одиночку мог написать его за разумное время. Таким

образом на большинстве работающих компьютеров появилась та или иная реализация стека и он стал стандартом де-факто сети Интернет.

В чём различие моделей OSI и TCP/IP? Обе они следуют принципу разделения задач, связанных с передачей данных, по нескольким уровням иерархии протоколов. Но в TCP/IP число этих уровней меньше: четыре против семи в модели OSI. Таблица 4-1 демонстрирует соответствие этих уровней.

Таблица 4-1. Уровни моделей OSI и TCP/IP

Уровень	OSI	TCP/IP
7	Прикладной (Application)	Прикладной (Application)
6	Представления (Presentation)	
5	Сеансовый (Session)	
4	Транспортный (Transport)	Транспортный (Transport)
3	Сетевой (Network)	Межсетевой (Internet)
2	Канальный (Data Link)	Канальный (Link)
1	Физический (Physical)	

Рассмотрим все уровни TCP/IP на примере реального сетевого пакета. Для этого воспользуемся анализатором трафика Wireshark. Скачайте и установите его на свой компьютер. После этого загрузите с Wiki ресурса Wireshark лог-файл с примером перехваченного Интернет-трафика. Откройте лог-файл `http.cap` в Wireshark. Диалог открытия файла можно вызвать по комбинации клавиш Ctrl+O. После этого окно анализатора должно выглядеть как на иллюстрации 4-4.

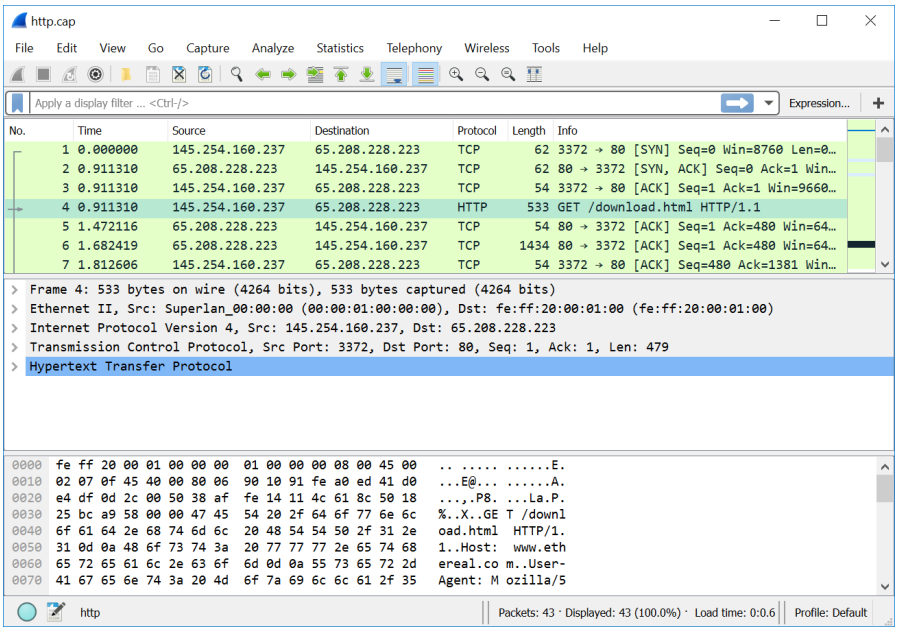


Иллюстрация 4-4. IP-пакеты в окне Wireshark

Окно анализатора разделено на три части. Верхняя из них представляет собой таблицу. Её горизонтальные ряды – это список перехваченных пакетов. Для каждого пакета в вертикальных столбцах приведена общая информация: адреса отправителя и получателя, время перехвата и т.д. Вы можете пролистать таблицу вниз и выбрать нужный пакет для вывода более подробной информации. Она отображается в средней части окна приложения. Здесь представлены **заголовки** всех протоколов, которые смог распознать Wireshark в этом пакете. Если вы выделите левым щелчком мыши один из заголовков, Wireshark подсветит соответствующие ему байты в нижней части окна. Более подробно интерфейс анализатора описан в официальной документации.

Мы рассмотрим пакет под номером четыре в лог-файле `http.cap`. Это типичный запрос браузера на загрузку веб-страницы из Интернета. Согласно таблице 4-1, в самом низу стека TCP/IP находятся протоколы канального уровня. Они отвечают за передачу пакетов по локальной сети. Как вы помните, в этом случае для обмена пакетами отправитель и получатель должны знать MAC-адреса друг друга. Этой информации будет достаточно для сетевого коммутатора, чтобы перенаправить пакет по назначению.

Согласно информации от Wireshark, отправитель четвёртого пакета в логе использует Ethernet II в качестве протокола канального уровня. Его заголовок идёт сразу после строчки “Frame” (кадр) в средней части окна анализатора. Если развернуть этот заголовок левым щелчком мыши по треугольнику рядом с ним, Wireshark отобразит содержащуюся в нём информацию: MAC-адреса получателя и отправителя. Кроме них, есть поле “Type” (тип) размером два байта. Оно содержит идентификатор протокола следующего уровня, который использовал отправитель. Заголовки протоколов следуют друг за другом в пакете. При этом можно рассматривать каждый протокол как контейнер содержащий заголовок и данные протокола следующего уровня. В нашем случае поле “Type” равно 0x0800, что соответствует **протоколу IP версии 4** (Internet Protocol Version 4 или IPv4).

IPv4 соответствует межсетевому уровню модели TCP/IP. Он отвечает за маршрутизацию пакетов между сетями. Самая важная информация его заголовка – это IP-адреса отправителя и получателя. Основываясь на них, маршрутизатор выбирает целевую сеть для передачи пакета. Чтобы прочитать эти адреса, разверните заголовок “Internet Protocol Version 4”. Кроме них есть несколько полей, информация которых также нужна для корректной маршрутизации. Например, поле “Time to live” (время жизни) определяет максимальное время, в течение которого пакет может передаваться по сети. Если оно оказалось превышено, первый маршрутизатор, получивший такой пакет, заблокирует его. Поля “Identification” (идентификатор) и “Fragment offset” (смещение фрагмента) хранят информацию, необходимую для механизма фрагментации. Он позволяет делить пакет на части (называемые фрагментами) и передавать их по отдельности через сеть. Такое разделение позволяет балансировать нагрузку в сети. Передача слишком больших пакетов увеличивает цену ошибки. Если один бит данных окажется искажённым в процессе передачи, весь пакет придётся отправлять повторно. Маленькие пакеты приводят к увеличению накладных расходов, т.е. уменьшится отношение полезной нагрузки (передаваемые данные) к служебной информации (заголовки протоколов). Последнее поле IPv4-заголовка называется “Protocol” (протокол). В нём хранится идентификатор протокола следующего уровня. В нашем случае – это **протокол управления передачей** (Transmission Control Protocol или TCP).

Протоколы транспортного уровня обеспечивают соединение между взаимодействующими по сети процессами, запущенными на разных хостах. Самая важная информация для этого соединения – номера портов, которые позволяют идентифицировать процессы отправителя и получателя. Разверните заголовок “Transmission Control Protocol” в окне Wireshark, чтобы прочитать значения “Source Port” (порт отправителя в нашем случае равен 3372) и “Destination Port” (порт получателя – 80). Кроме них, в заголовке есть поля “Sequence number” (порядковый номер) и “Acknowledgment number” (номер подтверждения). Эти номера нужны для **установки соединения** и обнаружения потерянных пакетов.

Сегодня в сети Интернет чаще других встречаются два протокола транспортного уровня: TCP и **протокол пользовательских датаграмм** (User Datagram Protocol или UDP). Основное различие между ними заключается в надёжности передачи данных. Протокол TCP имеет механизм проверки того, что все отправленные пакеты дошли до получателя. Если какой-то пакет был потерян, получатель просит его передать повторно. В протоколе UDP такого механизма нет. Получатель не проверяет последовательность входящих пакетов, а просто игнорирует потери.

Зачем может понадобиться такой ненадёжный протокол, как UDP? Наряду со всеми достоинствами у протокола TCP есть один существенный недостаток. Механизм обнаружения потерянных пакетов может привести к задержкам в передаче пакетов. В некоторых случаях такие задержки неприемлемы.

Для примера рассмотрим отправку и получение по сети видеопотока. В этом случае потеря одного кадра несущественна, поскольку воспроизведение видео можно продолжить со следующего. Однако, если мы используем протокол TCP, ОС запросит повторную отправку потерянного сетевого пакета. Тогда отправитель вместо следующего кадра будет пересылать потерянный. Это приведёт к остановкам при воспроизведении видео, поскольку у приложения видеопроигрывателя не будет нужного в данный момент кадра. Если же для передачи видеопотока применить протокол UDP, зависаний удастся избежать. При этом очень вероятно, что пользователь вообще не заметит потерянные кадры.

На самом верхнем уровне модели TCP/IP находятся прикладные протоколы. Их формат произволен, и разработчики программ могут выбирать его по своему усмотрению. Таким образом, порядок и значение байтов этой части сетевых пакетов целиком зависит от взаимодействующих приложений.

В нашем примере в качестве прикладного протокола используется **протокол передачи гипертекста** (Hypertext Transfer Protocol или HTTP). Данные этого протокола передаются в виде текста, который можно прочитать в нижней части окна Wireshark. Хост-отправитель пакета запрашивает у веб-сервера с **единым указателем ресурса** (Uniform Resource Locator или URL) “www.ethereal.com” страницу под названием “download.html”. URL, также известный как веб-адрес, представляет собой псевдоним для IP-адреса. Он был введён в употребление, чтобы упростить использование **всемирной паутиной** (World Wide Web или WWW). Благодаря URL пользователям нужно запоминать не IP-адреса, а названия сайтов.

Перехват трафика

Мы рассмотрели протоколы, используемые в сети Интернет. Теперь познакомимся с методом перехвата сетевого трафика двух взаимодействующих процессов, работающих на разных хостах. Анализ трафика игрового приложения – первый шаг при разработке внеигрового бота.

Тестовое приложение

Для начала напишем простое приложение, которое передаёт по сети несколько байтов. Оно состоит из двух частей: клиент и сервер. Благодаря интерфейсу loopback мы можем запустить их на одном компьютере и сымитировать передачу данных по сети. С помощью Wireshark перехватим этот трафик.

Перед тем как начать писать код, рассмотрим ресурс операционной системы, известный как **сетевой сокет** (network socket). Именно он предоставляет приложению функции ОС для передачи сетевых пакетов.

Понятие сокета тесно связано с портом и IP-адресом. Как вы помните, порты отправителя и получателя указаны в заголовках протоколов TCP и

UDP. Благодаря им ОС доставляет пакет тому процессу, который его ожидает.

Предположим, вы запускаете игровой клиент и чат-программу на своём компьютере. Что произойдёт если оба приложения решат использовать один и тот же сетевой порт для связи со своими серверами? В теории, каждая программа может выбрать порт по своему усмотрению. Чтобы предотвратить конфликты такого выбора, будет разумно зарезервировать некоторые порты для широко распространённых приложений. Это решение уже существует. Есть три диапазона портов:

1. **Общеизвестные** или системные от 0 до 1023. Эти порты используются процессами ОС, которые предоставляют широко распространённые сетевые сервисы.
2. **Зарегистрированные** или пользовательские от 1024 до 49151. Они частично зарезервированы за конкретными приложениями и сервисами **администрацией адресного пространства Интернет** (IANA).
3. **Динамические** или частные от 49152 до 65535. Представляют собой незарезервированные порты, которые могут быть использованы для любых целей.

Очевидно, что кто-то должен контролировать использование портов запущенными приложениями. Эту функцию выполняет ОС. Когда процесс хочет воспользоваться конкретным портом, он запрашивает у ОС сетевой сокет. Сокет – это абстрактный объект, представляющий собой конечную точку сетевого соединения. Этот объект содержит следующую информацию: IP-адрес, номер порта, состояние соединения. Как правило, приложение владеет сокетом и использует его монопольно. Когда он становится не нужен, его освобождают (release).

Вид сокета зависит от комбинации используемых протоколов. В наших примерах мы будем применять только пары: IPv4 и TCP, IPv4 и UDP.

Наше первое приложение отправляет один пакет данных по протоколу TCP. Оно состоит из двух Python-скриптов: `TestTcpReceiver.py` (см. листинг 4-1) и `TestTcpSender.py` (см. листинг 4-2). Алгоритм их работы следующий:

1. Скрипт `TestTcpReceiver.py` запускается первый. Он создаёт TCP-сокет, привязанный (bind) к порту 24000 и IP-адресу 127.0.0.1, известному как localhost (локальный хост). Такая конфигурация называется **TCP-сокет сервера**.
2. Скрипт `TestTcpReceiver.py` запускает цикл ожидания запроса на установку соединения через открытый им сокет. Говорят, что скрипт **слушает** (listen) порт 24000.
3. Запускается скрипт `TestTcpSender.py`. Он открывает TCP-сокет, но не привязывает его к какому-либо порту или IP-адресу. Эта конфигурация называется **TCP-сокет клиента**.
4. Скрипт `TestTcpSender.py` устанавливает соединение с сокетом получателя по IP-адресу 127.0.0.1 и порту 24000. После этого он отправляет пакет данных. ОС самостоятельно выбирает IP-адрес и порт отправителя, т.е. скрипт `TestTcpSender.py` не может выбрать их по своему усмотрению. После отправки пакета, скрипт освобождает свой сокет.
5. Скрипт `TestTcpReceiver.py` принимает запрос от отправителя на установку соединения, получает пакет данных, выводит их в консоль и освобождает свой сокет.

Рассмотренный нами алгоритм выглядит простым и прямолинейным. Однако, некоторые шаги по установке и разрыву TCP-соединения скрыты от пользователя и выполняются ОС автоматически. Мы увидим их, если перехватим и просмотрим трафик приложения в Wireshark.

Листинг 4-1. Скрипт `TestTcpReceiver.py`

```
import socket

def main():
  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
  s.bind(("127.0.0.1", 24000))
  s.listen(1)
  conn, addr = s.accept()
  data = conn.recv(1024, socket.MSG_WAITALL)
  print(data)
  s.close()

if __name__ == '__main__':
  main()
```

Скрипт `TestTcpReceiver.py` использует модуль `socket`, который предоставляет доступ к сокетам ОС. Алгоритм скрипта реализован в функции `main`. Рассмотрим её подробнее. Сначала вызывается функция

`socket` модуля `socket`. Она создаёт новый объект для сокета. У неё есть три входных параметра:

1. Набор протоколов, который будет использован при установке соединения. Наиболее часто выбираемые варианты:
* `AF_INET` (IPv4)
* `AF_INET6` (IPv6)
* `AF_UNIX` (локальное соединение)

2. Тип сокета. Может быть одним из следующих вариантов:
* `SOCK_STREAM` (TCP)
* `SOCK_DGRAM` (UDP)
* `SOCK_RAW` (без указания протокола транспортного уровня)

3. Номер протокола. Он используется, когда для указанного набора протоколов и типа сокета возможны несколько вариантов. В большинстве случаев этот параметр равен 0.

Мы создали сокет, использующий протоколы IPv4 и TCP, а затем поместили его в переменную с именем `s`. Следующий шаг нашего скрипта – привязать сокет к конкретному IP-адресу и порту с помощью метода `bind` объекта `s`. Затем с помощью метода `listen` запускаем цикл ожидания входящего соединения. Единственный входной параметр `listen` определяет максимальное число попыток установить соединение. В этой точке скрипт `TestTcpReceiver.py` останавливает своё выполнение, потому что вызов `listen` не возвращает управление, пока соединение не установлено.

Когда скрипт `TestTcpSender.py` пытается установить соединение, `TestTcpReceiver.py` принимает его через вызов метода `accept`. Этот метод возвращает два значения: объект соединения, пару IP-адрес и порт отправителя. Мы сохраняем их в переменные `conn` и `addr` соответственно. Для чтения данных из принятого пакета мы вызываем метод `recv` объекта `conn`. Затем печатаем их на консоль с помощью функции `print`.

Последним действием функции `main` освобождаем сокет через вызов его метода `close`. После этого ОС помечает ресурс как свободный. Теперь другое приложение может слушать TCP порт 24000.

Листинг 4-2 демонстрирует реализацию скрипта `TestTcpSender.py`.

Листинг 4-2. Скрипт `TestTcpSender.py`

```
import socket

def main():
  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
  s.settimeout(2)
  s.connect(("127.0.0.1", 24000))
  s.send(bytes([44, 55, 66]))
  s.close()

if __name__ == '__main__':
  main()
```

Здесь мы создаём такой же объект `s` для сокета, использующего протоколы IPv4 и TCP. Затем через метод `settimeout` устанавливаем двухсекундный тайм-аут на все операции с сокетом. Если сервер не ответит в течение этого времени на любой запрос клиента, будет сгенерировано исключение. Оно не обрабатывается в нашем скрипте, поэтому просто приведёт к его завершению.

Следующий шаг – установка соединения через вызов метода `connect`. В качестве входного параметра он получает пару: IP-адрес и порт сервера. В Python для объединения двух значений в пару используются круглые скобки. Метод `connect` возвращает управление сразу после успешной установки соединения. Теперь мы готовы к отправке пакета с данными. Для этого вызываем метод `send`. В примере отправляются три байта со значениями: 44, 55, 66. В конце функции `main` освобождаем сокет.

Перед запуском примера, необходимо проверить IP-адрес вашего интерфейса loopback. Для этого выполните следующие шаги:

1. Откройте окно “Network Connections” (сетевые подключения).
2. Правым щелчком мыши по иконке “Microsoft Loopback Adapter” откройте всплывающее меню и выберите пункт “Status” (состояние).
3. Нажмите кнопку “Details…” (сведения). Откроется окно “Network Connection Details” (сведения о сетевом подключении), в котором указан IPv4-адрес.

Если этот адрес отличается от 127.0.0.1, добавьте его в оба скрипта. В `TestTcpReceiver.py` нужно поправить вызов метода `bind`, а в `TestTcpSender.py` – вызов `connect`.

Лучше запускать оба скрипта в командной строке. Тогда вы сможете прочитать их выводы. Получатель должен напечатать три байта,

переданных через интерфейс loopback.

Перехват пакета

Перехватим и проанализируем трафик нашего тестового приложения с помощью Wireshark. Для этого выполните следующие действия:

1. Запустите Wireshark. В главном окне анализатора отобразится список сетевых интерфейсов, как на иллюстрации 4-5.

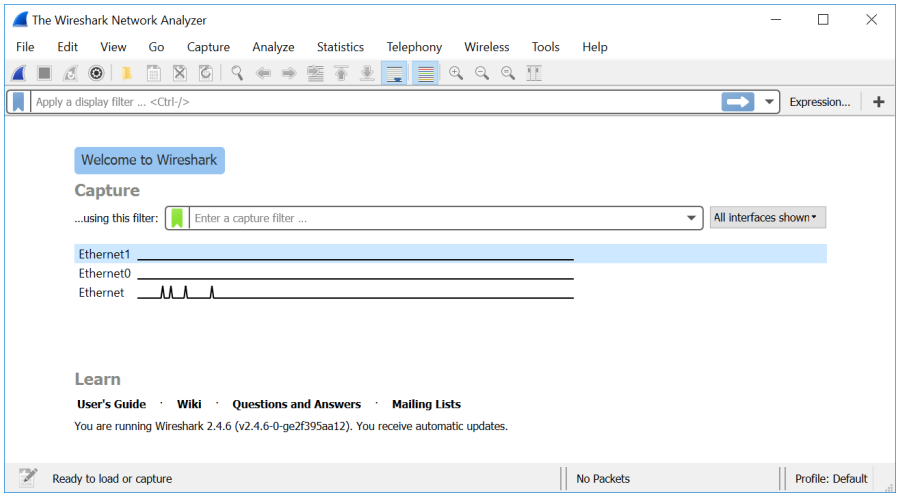


Иллюстрация 4-5. Список активных сетевых интерфейсов в окне Wireshark

2. Двойным щелчком левой кнопки мыши выберите интерфейс loopback в списке. Его имя вы можете уточнить в окне “Network Connections” (сетевые подключения). После выбора интерфейса, Wireshark сразу начнёт перехватывать проходящие через него пакеты.
3. Запустите скрипт `TestTcpReceiver.py`.
4. Запустите скрипт `TestTcpSender.py`.

В окне Wireshark вы увидите список перехваченных пакетов, как на иллюстрации 4-6.

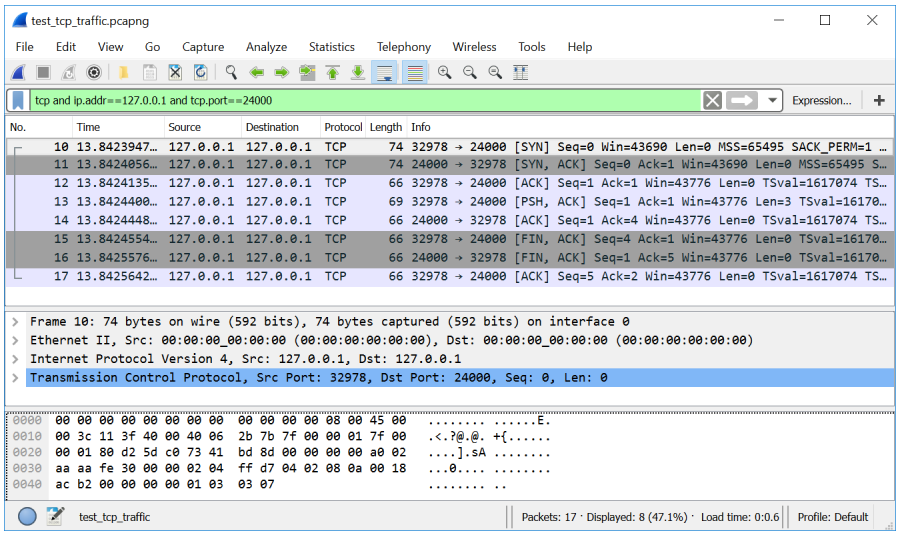


Иллюстрация 4-6. Перехваченные пакеты тестового приложения

Как правило, перехват трафика на сетевом интерфейсе нужен, чтобы отследить работу одного конкретного приложения. К сожалению, этим интерфейсом в то же самое время могут пользоваться сервисы ОС (например, менеджер обновлений) и другие приложения (например, браузер). Их пакеты также попадут в список перехваченных Wireshark. Чтобы исключить их из списка, анализатор предоставляет возможность фильтрации.

Под панелью иконок находится строка для ввода текста. Когда она пустая, в ней выводится серый текст: “Apply a display filter …” (применить фильтр отображения). В эту строку вы можете ввести правила фильтрации пакетов. Чтобы применить правила, нажмите иконку в виде стрелки слева от кнопки “Expression…” (выражение). После этого в списке перехваченных пакетов отобразятся только те, которые удовлетворяют условиям фильтрации.

Чтобы отобразить только пакеты нашего тестового приложения, применим следующий фильтр:

```
1 tcp and ip.addr==127.0.0.1 and tcp.port==24000
```

Он состоит из трёх условий. Первое из них представляет собой единственное слово “tcp”. Оно означает, что следует отобразить только пакеты, использующие протокол TCP. Второе условие “ip.addr==127.0.0.1”

проверяет IP-адрес отправителя и получателя. Если любой из них равен 127.0.0.1, пакет попадёт в список для отображения. Последнее условие “tcp.port==24000” ограничивает TCP-порты отправителя и получателя. Пакет будет отображён, если любой из них равен 24000.

Для комбинации правил в единый фильтр используется служебное слово “and” (И). Оно означает, что отобразятся пакеты, для которых выполняются все три условия одновременно. Другие часто используемые служебные слова: “or” (ИЛИ) и “not” (НЕ). Первое означает, что пакет будет отображён если хотя бы одно из указанных условий выполнено. Второе слово инвертирует условие. Служебные слова подробно описаны в официальной документации.

Указывать правила для фильтрации пакетов можно двумя способами: набирать текст условий в поле ввода (как мы сделали ранее) либо использовать диалог “Display Filter Expression” (фильтр отображения), приведённый на иллюстрации 4-7. Чтобы его открыть, нажмите кнопку “Expression…”.

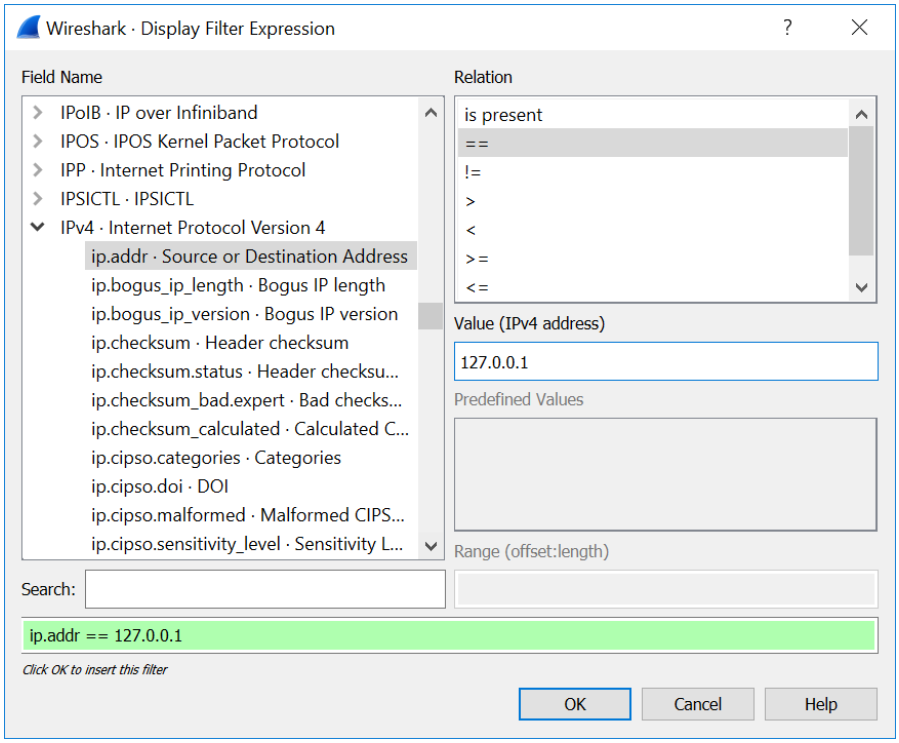


Иллюстрация 4-7. Диалог Display Filter Expression

В левой части диалога находится список “Field Name” (название поля) всех поддерживаемых протоколов и полей их заголовков. В списке “Relation” (отношение) приведены операторы отношения, с помощью которых вы можете накладывать ограничения на значения полей. Под ним находится поле ввода “Value” (значение), в котором указывается значение для сравнения. В нижней части диалога есть поле с получившимися правилами фильтрации в текстовой форме. На иллюстрации 4-7 это поле подсвечено зелёным цветом. Если в фильтре ошибка, цвет поменяется на красный.

Механизм фильтрации – это мощный инструмент, помогающий анализировать лог-файлы с перехваченным трафиком. Используйте его как можно чаще, чтобы ускорить свою работу с Wireshark.

Вернёмся к перехваченным пакетам нашего тестового приложения на иллюстрации 4-6. Почему в списке оказалось восемь пакетов, хотя наше приложение посылает один? Передача данных происходит только в пакете номер 13. Остальные, переданные до него (с номерами 10, 11, 12), нужны, чтобы установить TCP-соединение. Этот процесс известен как **тройное рукопожатие** (three-way handshake). Он состоит из следующих шагов:

1. Клиент (скрипт `TestTcpSender.py`) отправляет первый пакет (номер 10) на сервер. В TCP-заголовке этого пакета установлен флаг SYN, а sequence number или seq (порядковый номер) равен 0. Это означает, что клиент хочет установить соединение. Следующий фильтр отобразит в окне Wireshark только SYN пакеты:

```
1 tcp.flags.syn==1 and tcp.seq==0 and tcp.ack==0
```

2. Сервер (скрипт `TestTcpReceiver.py`) отвечает пакетом номер 11, в котором установлены флаги SYN и ACK. Кроме них в пакете передаётся acknowledgment number или ack (номер подтверждения), равный seq, полученный от клиента, плюс один. Таким образом подтверждается seq клиента. Также сервер передаёт клиенту собственный seq, равный 0. Чтобы отобразить ответы сервера на установку соединения, используйте следующий фильтр:

```
1 tcp.flags.syn==1 and tcp.flags.ack==1 and tcp.seq==0 and tcp.ack==1
```

3. Клиент отвечает пакетом номер 12 с установленным флагом ACK. Его ack-номер, равный единице, подтверждает seq сервера. После этого шага обе стороны подтвердили свои seq номера и готовы к взаимодействию. Следующий фильтр отображает ответ клиента:

```
1 tcp.flags.syn==0 and tcp.flags.ack==1 and tcp.flags.push==0 and tcp.seq==1 and tcp.a\
2 ck==1
```

Подробнее состояния клиента и сервера в процессе установки соединения рассмотрены в следующей статье.

Возможно, вы заметили, что в последнем фильтре для ответа клиента мы проверяем значение флага PUSH. Если этот флаг установлен в единицу, пакет содержит данные, отправленные приложением. Вы можете инвертировать условие, чтобы отобразить только эти пакеты:

```
1 not tcp.flags.push==0
```

Если вы хотите прочитать данные, отправленные нашим тестовым приложением, выделите пакет под номером 13 с установленным в единицу флагом PUSH. Затем щёлкните левой кнопкой мыши по пункту “Data” (данные) в списке заголовков. В результате в нижней части окна Wireshark синим цветом будут выделены соответствующие байты пакета, как на иллюстрации 4-8.

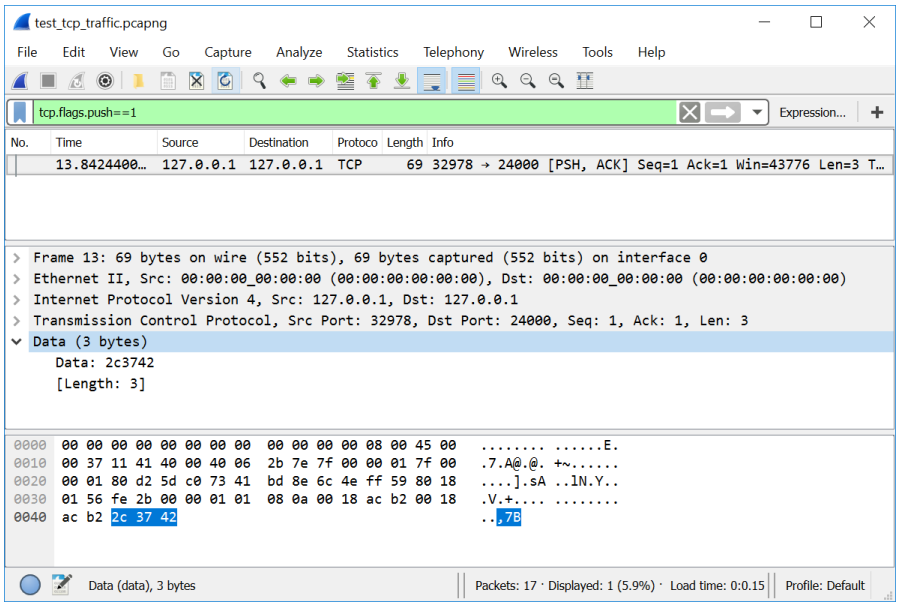


Иллюстрация 4-8. Пакет с данными

Тестовое приложение передаёт три байта, которые в шестнадцатеричной системе равны 2C, 37, 42. Если перевести эти числа в десятичную систему, получим: 44, 55, 66. Вы можете удостовериться в листинге 4-2, что именно эти три байта передаёт скрипт `TestTcpSender.py`.

Вы могли заметить на иллюстрации 4-6, что пакет с номером 14, следующий за передачей данных, имеет ack-номер равный четырём. Что означает это число? После установки соединения номера seq и ack используются для подтверждения числа байтов данных, полученных сервером от клиента. Следовательно, когда сервер получает данные, он отвечает пакетом с ack-номером, рассчитанным по формуле:

```
1 ack ответа = seq клиента + размер данных
```

В случае 14-ого пакета из нашего лог-файла, расчёт номера ack выглядит следующим образом:

```
1 ack = 1 + 3 = 4
```

Номер seq для этой формулы можно уточнить в последнем отправленном клиентом пакете с установленным флагом PUSH. В нашем случае это пакет с номером 13.

Иллюстрация 4-9 демонстрирует пример, когда клиент передаёт не один пакет данных, а несколько. В столбце `Info` вы можете проследить увеличение номеров ack и seq. Каждый пакет с подтверждением от сервера имеет ack, рассчитанный по рассмотренной выше формуле.

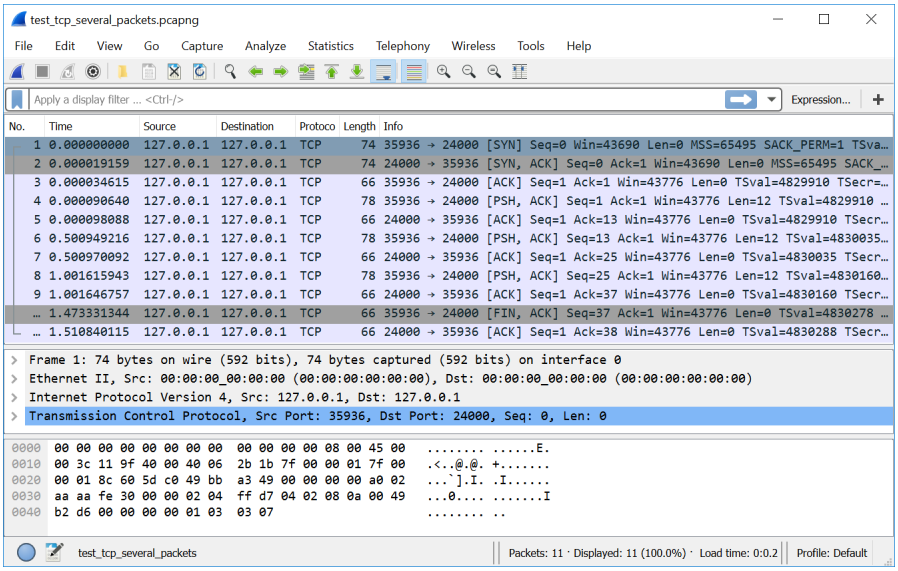


Иллюстрация 4-9. Последовательность TCP-пакетов

Обратите внимание, что клиент всегда посылает свои пакеты на целевой порт 24000. Порт отправителя равен 35936 на иллюстрации 4-9 и 32978 на иллюстрации 4-6. Как вы помните, ОС назначает его клиенту каждый раз, когда тот пытается установить новое соединение. Номер порта выбирается случайным образом, и его невозможно предсказать. Поэтому в условиях фильтрации пакетов лучше всегда проверять порт TCP-сервера, а не клиента.

Вернёмся к иллюстрации 4-6, на которой приведён TCP-трафик для передачи одного пакета данных. После его получения сервер отправляет пакет номер 14 с подтверждением. Затем следуют три пакета с номерами 15, 16 и 17 для закрытия TCP-соединения:

1. Клиент отправляет пакет номер 15, в котором установлен флаг FIN. Таким образом он запрашивает разрыв соединения. В нашем случае в этом пакете также установлен флаг ACK. С его помощью клиент подтверждает получение от сервера пакета номер 14, seq которого равен 1. Чтобы отобразить только этот пакет в Wireshark, примените следующий фильтр:

```
1 tcp.flags.fin==1 and tcp.dstport==24000
```

2. Сервер отвечает пакетом номер 16, в котором установлены флаги FIN и ACK. Его номер ack равен пяти, т.е. номеру seq клиента плюс один. Теперь флаг ACK означает, что сервер подтверждает получение FIN пакета. С помощью флага FIN сервер просит клиента закрыть соединение на своей стороне. Фильтр для отображения этого пакета следующий:

```
1 tcp.flags.fin==1 and tcp.srcport==24000
```

3. Клиент отвечает пакетом номер 17 с установленным флагом ACK. Он подтверждает получение запроса сервера на закрытие соединения. Номер seq этого пакета равен номеру ack последнего пакета (номер 16) от сервера. Фильтр для отображения:

```
1 tcp.flags.ack==1 and tcp.seq==5 and tcp.dstport==24000
```

Обратите внимание, что в этом фильтре мы проверяем номер seq пакета для того, чтобы найти последний пакет от клиента с установленным флагом ACK.

Подробнее закрытие TCP-соединения рассмотрено в статье.

UDP-соединение

Мы рассмотрели тестовое приложение, которое передаёт данные по протоколу TCP. Познакомились с основными принципами его работы и знаем как перехватить и проанализировать такой вид трафика. Однако, многие онлайн-игры используют протокол UDP вместо TCP.

Перепишем наше тестовое приложение так, чтобы оно использовало протокол UDP. В этом случае его алгоритм будет выглядеть следующим образом:

1. Скрипт `TestUdpReceiver.py` (из листинга 4-3) запускается первым. Он открывает UDP-сокет и привязывает (bind) его к порту 24000 и IP-адресу 127.0.0.1. UDP-сокеты, в отличие от TCP, равноправны. Это значит, что любой из них может отправлять данные в произвольный момент времени. Процедур установки и разрыва соединения нет.

2. Скрипт `TestUdpReceiver.py` ожидает входящего пакета от отправителя.
3. Скрипт `TestUdpSender.py` (из листинга 4-4) запускается вторым. Он открывает UDP-сокет и привязывает его к порту 24001 и адресу localhost. Последний шаг необязателен. Тогда ОС назначит произвольный порт отправителю UDP-пакетов. Однако, явная привязка к порту может быть полезной, если понадобится передавать данные в обоих направлениях.
4. Скрипт `TestUdpSender.py` отправляет пакет данных, после чего освобождает свой сокет.
5. Скрипт `TestUdpReceiver.py` получает пакет, выводит на консоль его содержимое и освобождает свой сокет.

Как видите, алгоритм тестового приложения стал проще, по сравнению с использованием протокола TCP. Нет необходимости устанавливать и разрывать соединение. Приложение только отправляет единственный пакет с данными.

Листинг 4-3. Скрипт `TestUdpReceiver.py`

```
import socket

def main():
  s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, 0)
  s.bind(("127.0.0.1", 24000))
  data, addr = s.recvfrom(1024, socket.MSG_WAITALL)
  print(data)
  s.close()

if __name__ == '__main__':
  main()
```

Алгоритм этого скрипта похож на `TestTcpReceiver.py`. В отличие от него, здесь нет цикла ожидания и установки соединения. В качестве типа сокета `s` указан `SOCK_DGRAM`, который соответствует протоколу UDP. Для получения пакета используется метод `recvfrom` объекта `s`. В отличие от метода `recv` TCP-сокета, он возвращает пару значений: принятые данные и IP-адрес отправителя. Поскольку для UDP не устанавливается соединения, мы не вызываем метод `accept`. Поэтому IP-адрес отправителя можно получить только через вызов `recvfrom`. Если этот адрес не важен, можно использовать метод `recv`, как и в случае TCP.

Листинг 4-4. Скрипт `TestUdpSender.py`

```
import socket

def main():
  s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, 0)
```

```
  s.bind(("127.0.0.1", 24001))
  s.sendto(bytes([44, 55, 66]), ("127.0.0.1", 24000))
  s.close()

if __name__
  main()
```

В скрипте `TestUdpSender.py` мы так же указываем тип сокета `SOCK_DGRAM` при создании его объекта `s`. Нам не нужен тайм-аут на операции с сокетом, поскольку протокол UDP не предполагает подтверждений для передаваемых пакетов. Вместо этого мы просто отправляем данные и освобождаем сокет.

Запустите Wireshark и начните перехват трафика на интерфейс loopback. После этого запустите скрипты `TestUdpReceiver.py` и `TestUdpSender.py`. Вы должны получить результат, приведённый на иллюстрации 4-10.

Если вы видите несколько пакетов в списке перехваченных Wireshark, примените следующий фильтр:

```
udp.port==24000
```

Вы увидите единственный пакет, содержащий три байта данных: 2C, 37, 42.

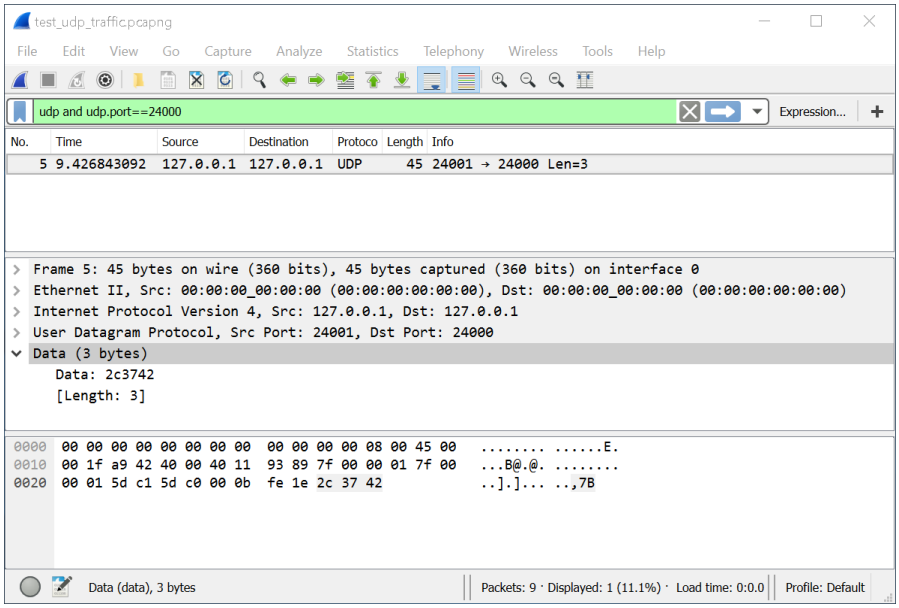


Иллюстрация 4-10. Перехваченный UDP-пакет

Пример бота для NetChess

Мы узнали достаточно, чтобы написать простого внеигрового бота. Он будет делать ходы в шахматной программе NetChess. Эта программа состоит из клиентской и серверной частей. Она позволяет играть двум пользователям по локальной сети. Вы можете бесплатно скачать её на сайте SourceForge. Чтобы установить игру, просто распакуйте архив с ней в любой каталог.

Рассмотрим интерфейс игры. Её главное окно изображено на иллюстрации 4-11. Большую его часть занимает шахматная доска с фигурами. Главное меню находится в верхней области окна. Ряд иконок под меню дублирует некоторые из его функций.

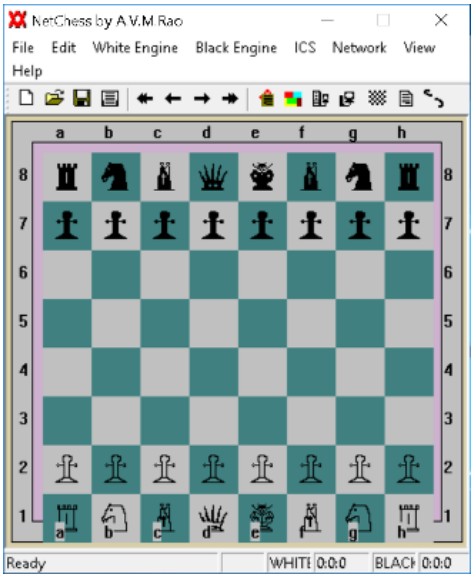


Иллюстрация 4-11. Окно NetChess

Чтобы начать игру, необходимо запустить приложение NetChess и назначить ему роль сервера. После этого второй игрок запускает приложение на другом компьютере и настраивает его на роль клиента. Он подключается к серверу, и игра начинается. Благодаря интерфейсу loopback мы можем запустить клиент и сервер на одном хосте.

Чтобы запустить NetChess и начать игру, выполните следующие действия:

1. Дважды запустите исполняемый файл `NetChess2.1.exe` из каталога `Debug` игры. В результате откроется два окна NetChess,

соответствующие двум процессам. Выберите, кто из них будет выполнять роль сервера.
2. Переключитесь на окно сервера и выберите пункт меню “Network” -> “Server” (“Сеть” -> “Сервер”). Откроется диалог конфигурации приложения в роли сервера, как на иллюстрации 4-12.

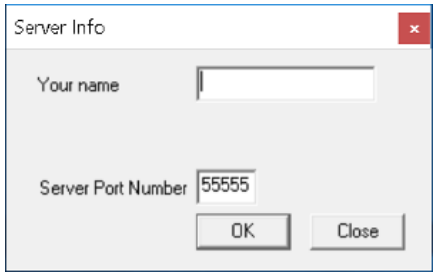


Иллюстрация 4-12. Диалог конфигурации сервера

3. Введите имя пользователя, который играет на стороне сервера, и нажмите кнопку “OK”.
4. Переключитесь на окно приложения NetChess, выполняющее роль клиента. Выберите пункт меню “Network” -> “Client” (“Сеть” -> “Клиент”). Откроется диалог конфигурации клиента, как на иллюстрации 4-13.

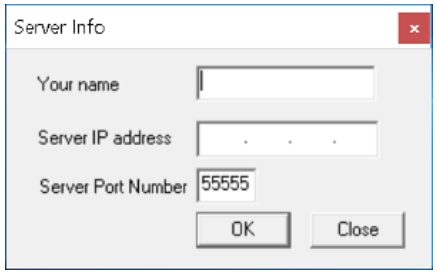


Иллюстрация 4-13. Диалог конфигурации клиента

5. Введите имя пользователя на стороне клиента и IP-адрес сервера (в моём случае это 169.254.144.77). Затем нажмите кнопку “OK”.
6. Переключитесь на окно сервера. Когда клиент попытается подключиться, должен открыться диалог “Accept” (принять), как на

иллюстрации 4-14. В нём выберите цвет фигур (чёрный, белый, случайный). После этого нажмите кнопку “Accept” (принять).

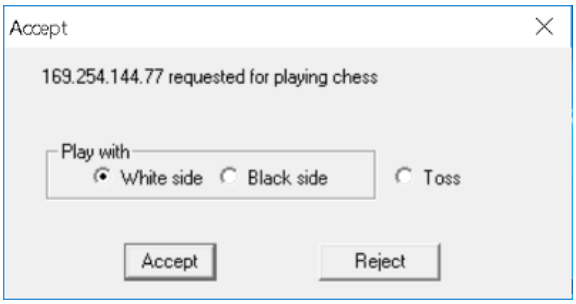


Иллюстрация 4-14. Диалог подключения клиента

7. Переключитесь на окно клиента. Вы увидите сообщение об успешном подключении к серверу. В нём выводится имя оппонента и цвет его фигур (см иллюстрацию 4-15).

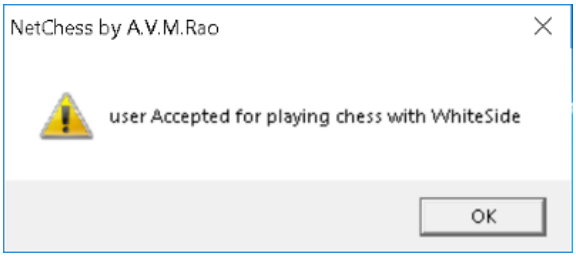


Иллюстрация 4-15. Диалог подтверждения подключения

8. Переключитесь на окно сервера и выберите пункт меню “Edit” -> “Manual Edit” -> “Start Editing” (“Редактирование” -> “Ручное редактирование” -> “Начать редактирование”). Откроется диалог с подтверждением, в котором вы должны нажать кнопку “Yes” (да). После этого приложение позволит вам запустить игровые часы.
9. Переключитесь на окно клиента и подтвердите включение режима “Manual Edit” в открывшемся диалоге. Для этого нажмите кнопку “Yes”.
10. Переключитесь на окно сервера. Вы увидите сообщение, что клиент подтвердил включение режима “Manual Edit”. Закройте его нажатием кнопки “OK”. Затем уберите галочку с пункта меню “Edit” -> “Manual

Edit” -> “Pause clock” (“Редактирование” -> “Ручное редактирование” -> “Остановить часы”).

Игровые часы запустятся, и белая сторона может сделать первый ход. Для этого достаточно перетащить мышкой нужную фигуру на другую клетку доски.

Обзор бота

Наш внеигровой бот будет подключаться к серверу и полностью замещать собой приложение NetChess, выполняющее роль клиента.

У бота есть много способов выбрать свой ход. Предлагаю остановиться на самом простом решении. Ведь мы рассматриваем взаимодействие с игровым сервером, а не алгоритмы шахматных программ. Наш бот будет зеркально повторять ходы игрока до тех пор, пока это позволяют правила игры. Задача выглядит достаточно простой, но потребует изучения протокола NetChess.

Приложение NetChess распространяется с открытым исходным кодом. Вы можете изучить код и быстро разобраться в протоколе приложения. Мы выберем другой путь. Давайте предположим, что NetChess – проприетарная игра и её исходный код недоступен. Для исследования у нас есть только перехваченный сетевой трафик между клиентом и сервером.

Изучение трафика NetChess

Мы рассмотрели шаги, необходимые для установки соединения между клиентом и сервером NetChess, а также чтобы начать игру. Теперь мы можем перехватить трафик и найти сетевые пакеты, соответствующие каждому из этих шагов. Но сначала рассмотрим два важных вопроса.

Как мы будем отличать трафик NetChess от остальных приложений в Wireshark логе? Если бы мы использовали сетевую плату вместо интерфейса loopback, в лог попали бы пакеты всех работающих в данный момент сетевых приложений. Но пакеты NetChess мы можем отличить по номеру порта. Мы указали его при настройке серверной части приложения. По умолчанию он равен 55555. Применим следующее условие проверки порта в качестве Wireshark фильтра:

```
1 tcp.port==55555
```

Теперь в логе будет выводиться только трафик NetChess.

Следующий вопрос: как именно следует перехватывать трафик? Можно просто запустить Wireshark, начать прослушивать интерфейс loopback и сыграть несколько игр подряд. Поступив так, мы потеряем важную информацию, которая очень пригодилась бы для изучения трафика. В Wireshark логе, собранном по нескольким играм, будет сложно различить отдельные ходы каждой стороны. Например, какой именно пакет соответствует первому ходу белых? В логе накопилось более ста пакетов, а мы не можем даже сказать, когда начиналась каждая игра. Чтобы избежать этого затруднения, будем проверять Wireshark лог сразу после каждого совершённого действия. В этом случае мы легко отличим соответствующие ему пакеты.

Теперь запустите Wireshark, NetChess клиент и сервер. Начните прослушивание интерфейса loopback в анализаторе. После этого выполните следующие действия:

1. Запустите NetChess в режиме сервера (настройка “Network” -> “Server”). После этого действия приложение только открывает сокет. Поэтому в логе Wireshark новых пакетов не появится.
2. Подключитесь клиентом NetChess к серверу (настройка “Network” -> “Client”). В Wireshark окне появятся три пакета, как на иллюстрации 4-16. Это установка TCP-соединения через тройное рукопожатие.

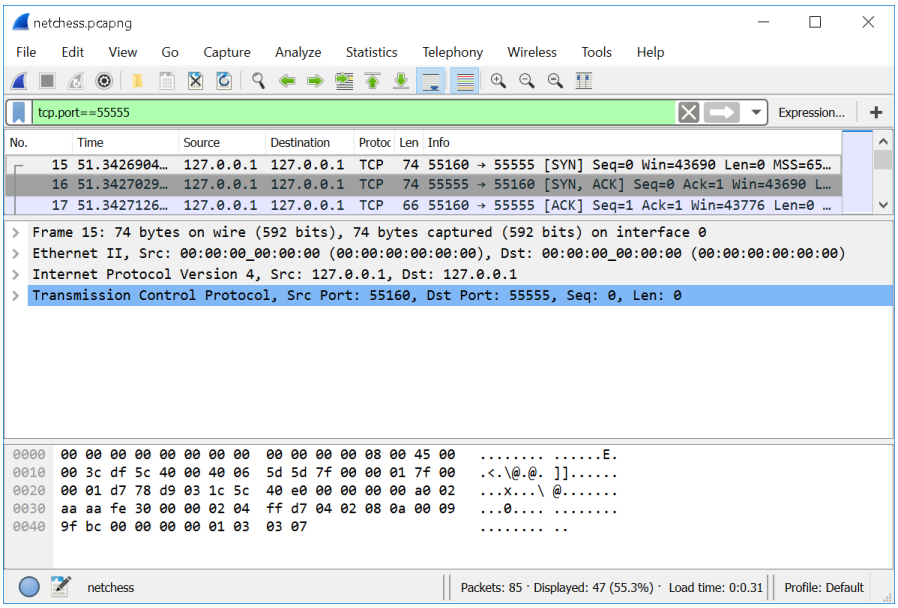


Иллюстрация 4-16. Установка соединения между клиентом NetChess и сервером

3. Сервер принимает соединение клиента. После этого анализатор перехватит два пакета, отправленные сервером. На иллюстрации 4-17 их номера 22 и 24. Клиент подтверждает их получение и сам посылает два пакета с данными (их номера 26 и 28).

Остановимся на этом шаге и рассмотрим только что перехваченные пакеты. Первый пакет от сервера под номером 22 содержит следующие данные:

```
1 0f 00 00 00
```

Попробуйте перезапустить клиент и сервер NetChess. После этого снова установите соединение между ними. Данные, передаваемые первым пакетом не изменятся. Вероятнее всего, на прикладном уровне модели TCP/IP они означают, что сервер принял соединение клиента. Чтобы проверить это предположение, попробуйте на стороне сервера отклонить подключение клиента. В этом случае данные пакета изменятся на следующие:

```
1 01 00 00 00
```

Из этого следует, что наша гипотеза верна. Приняв соединение, сервер отвечает первым байтом 0f. Иначе в ответе будет 01.

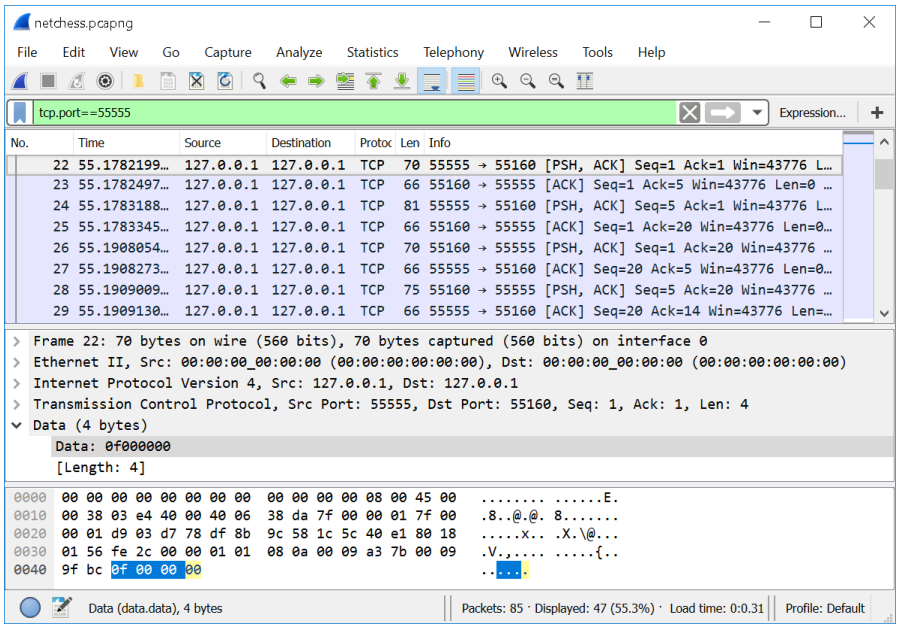


Иллюстрация 4-17. Подтверждение подключения NetChess сервером

Второй пакет от сервера с номером 24 содержит следующие байты данных:

```
1 0b 02 46 6d e7 5a 73 72 76 5f 75 73 65 72 00
```

В моём случае игрок на стороне сервера выбрал белые фигуры и ввёл имя “srv_user”. Wireshark способен частично декодировать эти данные. Согласно иллюстрации 4-18, байты с 7-ого по 15-ый соответствуют имени пользователя.

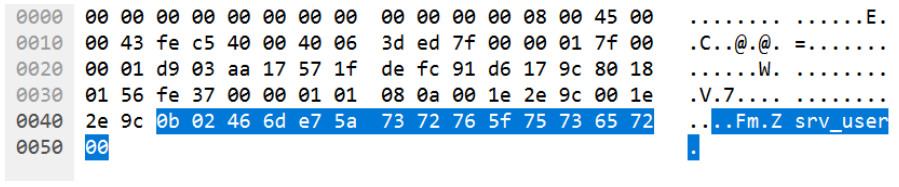


Иллюстрация 4-18. Декодирование данных второго пакета от сервера в Wireshark

Что означают первые шесть байтов в ответе сервера? Перезапустите приложение и заставьте его отправить этот пакет снова. Не забудьте выбрать то же имя пользователя “srv_user” и белые фигуры на стороне сервера. Благодаря этому уже известные нам байты данных не изменятся.

После перезапуска NetChess, у меня получились следующие данные в пакете:

```
1 0b 02 99 b3 ee 5a 73 72 76 5f 75 73 65 72 00
```

Обратите внимание, что первые два байта (0b и 02) не изменились. Скорее всего, в них закодирован цвет фигур, который выбрал игрок на стороне сервера. Попробуйте перезапустить NetChess и выбрать сторону чёрных. Данные этого пакета поменяются:

```
1 0b 01 ba 45 e8 5a 73 72 76 5f 75 73 65 72 00
```

Если повторить тест с выбором чёрных фигур несколько раз, второй байт всегда будет равен 01. Это подтверждает наше предположение. Цвет фигур игрока на стороне сервера кодируется согласно таблице 4-2. Эта информация может оказаться полезной для бота.

Таблица 4-2. Кодирование цвета фигур игрока на стороне сервера

Байт	Цвет
01	Чёрный
02	Белый

Следующие два пакета с данными отправляются клиентом. Первый из них под номером 26 содержит байты:

```
1 09 00 00 00
```

Они не изменятся, если мы перезапустим приложение и попробуем поменять имя игрока на стороне сервера или цвет его фигур. Поэтому предположительно это неизменный ответ клиента.

Следующий пакет под номером 28 содержит данные:

```
1 0c 63 6c 5f 75 73 65 72 00
```

Wireshark декодирует эти байты, начиная со второго, как имя игрока на стороне клиента (см. иллюстрацию 4-19). Значение первого байта неясно. Оно не меняется после перезапуска приложения. Бот может обращаться с ним как с константой и всегда включать в свой ответ серверу.

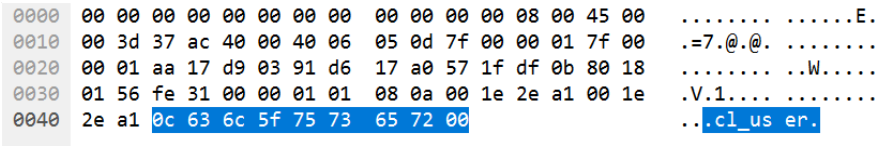


Иллюстрация 4-19. Декодирование данных второго пакета от клиента в Wireshark

Продолжим действия в приложении NetChess, необходимые для начала игры. Включим режим “Manual Edit” на стороне сервера (“Edit” -> “Manual Edit” -> “Start Editing”). После этого сервер отправляет два пакета клиенту.

Первый пакет под номером 41 на иллюстрации 4-20 содержит следующие данные:

```
1 0a 00 00 00
```

Вероятнее всего, первый байт 0a соответствует коду запроса сервера. Данные второго пакета под номером 43 выглядят так:

```
1 13 73 72 76 5f 75 73 65 72 00
```

Мы уже встречали набор байтов со 2-ого по 9-ый и знаем, что он соответствует строке “srv_user”. Первый же байт со значением 13 не меняется и наш бот может его игнорировать.

Когда клиент подтверждает включение режима “Manual Edit”, он отправляет два пакета с номерами 45 и 47 на иллюстрации 4-20. Их данные следующие:

```
1 01 00 00 00
2 17
```

При получении запроса сервера, начинающегося с 0a, бот должен повторить этот ответ без изменений.

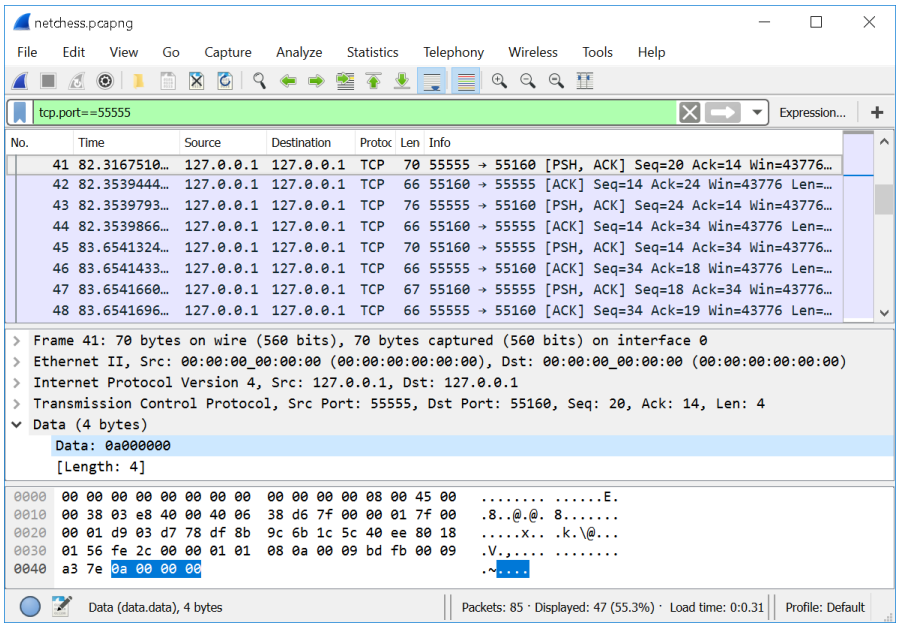


Иллюстрация 4-20. Включение режима Manual Edit сервером

Чтобы начать игру, нам осталось только включить часы. После этого действия сервер отправляет два пакета с номерами 54 и 56 на иллюстрации 4-21. Данные этих пакетов следующие:

```
1 02 00 00 00
2 22 00
```

Клиент не отвечает на эти пакеты, поэтому наш бот может их просто проигнорировать.

Все последующие пакеты (начиная с номера 58) передают данные о перемещении фигур игроками. Первой ходит белая сторона. В нашем случае это игрок на стороне сервера. Каждому ходу соответствует два пакета с данными в Wireshark логе.

Если белые сделают первый ход e2-e4, сервер передаст пакеты со следующими данными:

```
1 07 00 00 00
2 00 00 06 04 04 04 00
```

Попробуйте сделать ещё несколько ходов за обе стороны. Вы заметите, что данные первого из двух пакетов (07 00 00 00) не меняются. По ним бот

может определить, что передаётся ход игрока.

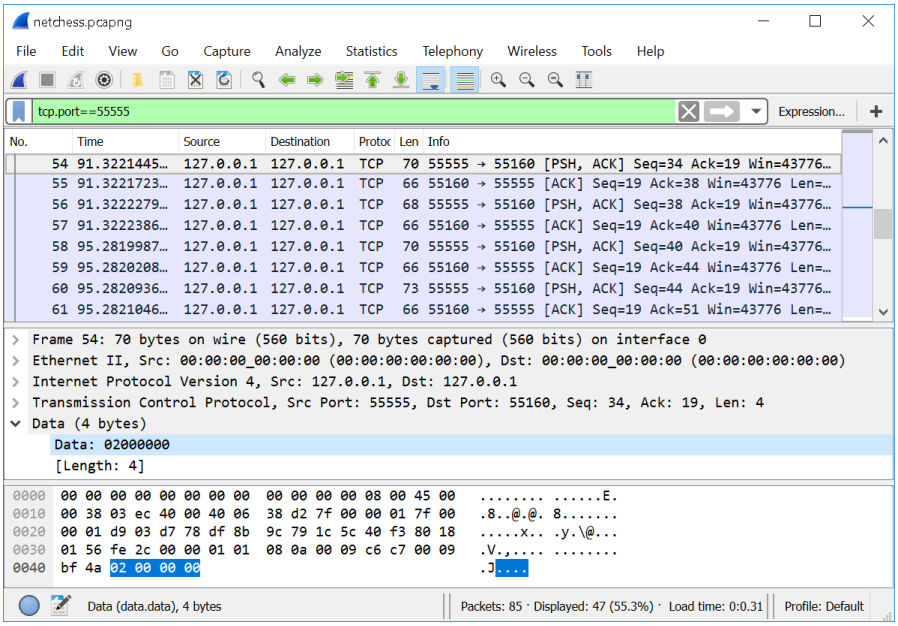


Иллюстрация 4-21. Запуск игровых часов сервером

Мы подошли к самому важному вопросу: как декодировать данные о ходе игрока? Представим себе шахматную доску. В ней всего 64 поля: 8 по вертикали и 8 по горизонтали. По вертикали поля нумеруются цифрами от 1 до 8, а по горизонтали – латинскими буквами от a до h. Очевидно, что ход каждого игрока должен содержать информацию о поле, где находится фигура сейчас, и поле, куда её следует переместить.

Вернёмся к перехваченному пакету с информацией о перемещении фигуры. Его данные содержат четыре ненулевых байта. Попробуйте сделать ещё несколько ходов. Первые два и последний байт всегда равны нулю, а остальные – нет. Следовательно, начальная и конечная позиция фигуры должна быть закодирована в этих четырёх байтах. То есть каждое поле задаётся двумя байтами.

Предположим, что первым указывается текущее поле фигуры. В нашем случае клетке e2 соответствуют два байта 06 04, а e4 соответствуют 04 04. Обратите внимание, что буква у обоих полей одинакова. Исходя из этого, предположим, что байт 04 соответствует букве “e”.

Теперь сделайте ход пешкой на поле с другой буквой, чтобы подтвердить наше предположение. В случае “d2-d4” данные соответствующего пакета выглядят следующим образом:

```
1 00 00 06 03 04 03 00
```

Получается, что букве “d” соответствует байт 03. Логично предположить, что коды букв идут последовательно один за другим. Учитывая это, составим таблицу 4-3 соответствия букв и их кодов.

Таблица 4-3. Коды букв полей шахматной доски

Байт	Буква
00	a
01	b
02	c
03	d
04	e
05	f
06	g
07	h

Как мы получили эту таблицу? Начнём заполнять её левый столбец с уже известных нам байтов 03 и 04, которые соответствуют буквам “d” и “e”. Затем продолжим вверх значения в левом столбце: 02, 01, 00. Точно так же продолжим вверх значения в правом столбце: “c”, “b”, “a”. Аналогично заполним строки таблицы после байта 04.

Теперь составим похожую таблицу для номеров клеток. Мы уже знаем, что байт 06 соответствует номеру 2, а 04 – номеру 4. Поместим эти значения в таблицу и заполним остальные её строки. Вы должны получить таблицу 4-4.

Таблица 4-4. Коды номеров полей шахматной доски

Байт	Номер
07	1
06	2
05	3
04	4
03	5
02	6
01	7
00	8

Проверьте наши выводы, делая различные игровые ходы. По номерам и буквам клеток вы легко сможете предсказать данные пакетов, которые отправляют друг другу клиент и сервер.

Теперь мы знаем об игровом протоколе всё необходимое, чтобы написать бота.

Реализация бота

Начало игры

Первая задача бота – подключиться к серверу и начать игру в качестве клиента. Мы подробно рассмотрели все пакеты, которыми обмениваются обе стороны на этом этапе. Теперь реализуем скрипт, отвечающий на запросы сервера точно так же, как клиент NetChess. Результат приведён в листинге 4-5.

Листинг 4-5. Скрипт `StartGameBot.py`

```
import socket

def main():
  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
  s.settimeout(60)
  s.connect(("127.0.0.1", 55555))

  # получить от сервера подтверждение соединения
  s.recv(1024, socket.MSG_WAITALL)
  s.recv(1024, socket.MSG_WAITALL)

  # отправить имя пользователя на стороне клиента
  s.send(bytes([0x09, 0, 0, 0]))
  s.send(bytes([0x0C, 0x63, 0x6C, 0x5F, 0x75, 0x73, 0x65, 0x72, 0x00]))

  # получить от сервера уведомление о включении режима "Manual Edit"
  s.recv(1024, socket.MSG_WAITALL)
```

```
  s.recv(1024, socket.MSG_WAITALL)

  # отправить подтверждение клиентом режима "Manual Edit"
  s.send(bytes([0x01, 0, 0, 0]))
  s.send(bytes([0x17]))

  # получить от сервера уведомление о включении игровых часов
  s.recv(1024, socket.MSG_WAITALL)
  s.recv(1024, socket.MSG_WAITALL)

  s.close()

if __name__ == '__main__':
  main()
```

Некоторые строки скрипта `StartGameBot.py` начинаются со знака решётки (#). Это комментарии, а не код.

Первые три строки функции `main` нам уже знакомы. Они устанавливают TCP-соединение. Обратите внимание, что мы указали 60 секундный тайм-аут для сокета. в течение этого времени вызовы `recv` ожидают пакеты от сервера. За это время игрок должен успеть сделать свой ход.

Затем идут два вызова `recv`, чтобы получить подтверждение от сервера об успешном соединении. В этих пакетах указано имя игрока и цвет его фигур. Эти данные не важны для бота, поэтому он их игнорирует.

Почему цвет фигур оппонента игнорируется ботом? На самом деле вопрос стоит сформулировать иначе: сможет ли бот сыграть любым цветом? Ответ – нет. Наш бот отвечает на ходы игрока зеркально, то есть повторяет их. Следовательно, он может сделать свой ход только после человека. То есть бот всегда играет за чёрных.

Получив подтверждение от сервера, бот отправляет имя пользователя на стороне клиента. Оно не важно. Для примера будем отправлять строку “cl_user”, которая в виде байтового массива представляется следующим образом:

```
63 6C 5F 75 73 65 72
```

Перед именем пользователя добавим обязательную константу 0c.

На следующем шаге сервер включает режим “Manual Edit”. Получив от него уведомление, бот отправляет пакет с подтверждением. После этого сервер запускает игровые часы. На это действие уведомление от клиента не требуется.

Можно ли удалить лишние `recv` вызовы из скрипта `StartGameBot.py`? В нём мы не используем данные пакетов, полученных от сервера. Другими словами бот игнорирует информацию о выбранном пользователем имени и цвете фигур, а также код режима “Manual Edit”. Всё, что на самом деле нужно боту, – это данные с ходами игрока. Да, мы могли бы удалить лишние вызовы `recv`, но тогда возникает проблема. Как бот выберет правильные моменты времени для отправки подтверждений на действия сервера? Можно останавливать выполнение скрипта с помощью функции `sleep` на время достаточное пользователю, чтобы напечатать своё имя или включить режим “Manual Edit”. Но такое решение ненадёжно. Если бот ответит раньше, чем сервер отправит ему запрос на подтверждение, порядок процедуры запуска игры нарушится. Получается, что единственный надёжный способ для бота вовремя реагировать на действия игрока – это получать все пакеты от сервера с помощью вызова `recv`. Далее зная заранее последовательность шагов для начала игры, бот может точно установить момент получения пакета с первым ходом пользователя.

Повторение ходов игрока

В листинге 4-5 мы рассмотрели часть скрипта бота, которая отвечает за процесс начала игры. После него пользователь делает свой первый ход, на который бот должен ответить. Реализуем алгоритм для зеркального повторения ходов игрока.

Как правильно выбрать фигуру для перемещения и её новое поле? Рассмотрим несколько примеров зеркальных ходов в таблице 4-5.

Таблица 4-5. Зеркальные ходы

Ход	Байты данных	Зеркальный ход	Байты данных
e2-e4	00 00 06 04 04 04 00	e7-e5	00 00 01 04 03 04 00
d2-d4	00 00 06 03 04 03 00	d7-d5	00 00 01 03 03 03 00
b1-c3	00 00 07 01 05 02 00	b8-c6	00 00 00 01 02 02 00

Первый ход в таблице e2-e4 делает белая пешка. Ему соответствует зеркальный ход чёрной пешкой e7-e5. Следующие ходы делают пешки на линии “d”. Затем идёт ход белого коня b1-c3. Прочитав соответствующий ему зеркальный ход чёрных, вы, возможно, заметите некоторые закономерности в байтах данных.

Первая закономерность связана с буквенными обозначениями. Предположим, что фигура, которая делает ход, находится на поле с буквой b. Тогда выполняющая зеркальный ход фигура тоже будет находиться на поле b. Буквы полей, в которые фигуры переместятся, также совпадут. Это правило выполняется для всех фигур.

Вторая закономерность поможет нам рассчитать номера клеток. Внимательно посмотрите на следующие пары чисел:

- 6 и 1
- 4 и 3
- 7 и 0
- 5 и 2

Как из правого числа получить левое? Для этого надо вычесть его из семи. Это правило выполняется для каждой из рассмотренных пар.

Теперь реализуем алгоритм расчёта зеркальных ходов. Результат приведён в листинге 4-6.

Листинг 4-6. Алгоритм расчёта зеркальных ходов

```
while(1):
  # получить от сервера ход игрока
  s.recv(1024, socket.MSG_WAITALL)
  data = s.recv(1024, socket.MSG_WAITALL)
  print(data)

  start_num = 7 - data[2]
  end_num = 7 - data[4]

  # отправить ход бота
  s.send(bytes([0x07, 0, 0, 0]))
  s.send(bytes([0, 0, start_num, data[3], end_num, data[5], 0x00]))
```

Алгоритм работает в бесконечном цикле `while`. Сначала мы получаем пакет от сервера с ходом игрока и сохраняем его данные в переменной `data`. С помощью функции `print` выводим эти данные на консоль. Далее вычисляем номер клетки чёрной фигуры, которая должна сделать ход. Для

расчёта используем третий байт массива `data` (с индексом 2). Он соответствует номеру начального поля белой фигуры. Результат сохраняем в переменной `start_num`. Аналогично вычисляем номер клетки, куда фигура должна походить. Результат сохраняем в переменной `end_num`. После этого отправляем два пакета с ходом бота. Первый из них содержит константные данные (07 00 00 00). Второй – рассчитанные номера клеток и те же буквы, что и в ходе игрока. Они хранятся в байтах с индексами 3 и 5 массива `data`.

Полная реализация бота доступна в файле `MirrorBot.py` из архива примеров к этой книге. В нём объединён код из листингов 4-5 и 4-6.

Чтобы протестировать бота, выполните следующие действия:

1. Запустите приложение NetChess.
2. Настройте его на работу в режиме сервера.
3. Запустите скрипт `MirrorBot.py`.
4. В приложении включите режим “Manual Edit”.
5. Запустите игровые часы.
6. Сделайте первый ход за белых.

Бот будет повторять каждый ваш ход до тех пор, пока это позволяют правила игры. Если такой ход невозможен, бот не будет ничего делать.

Выводы

Рассмотрим эффективность нашего внеигрового бота, сопоставив его достоинства и недостатки.

Достоинства бота:

1. Он получает полную и точную информацию о состоянии игровых объектов.
2. Он может симулировать действия игрока без каких-либо ограничений.

Недостатки бота:

1. Анализ протокола взаимодействия клиента и сервера требует времени. Чем сложнее игра, тем более трудоёмким становится этот

процесс.
2. Чтобы защититься от этого типа ботов, достаточно зашифровать трафик между клиентом и сервером.
3. Незначительные изменения в протоколе игры приводят к обнаружению бота. Также они могут помешать его работе, поскольку сервер, скорее всего, заблокирует пакеты устаревшего формата.

Мы можем обобщить наши выводы на большинство внутриигровых ботов. Они хорошо справляются с автоматизацией игрового процесса, но только до тех пор, пока на стороне сервера не поменяется протокол взаимодействия. После этого ваша игровая учётная запись будет заблокирована с большой вероятностью. Разработка ботов этого типа требует значительных усилий и времени.

Методы защиты от внеигровых ботов

Мы разработали бота для NetChess. Это простое приложение для игры в шахматы по локальной сети. Современные онлайн-игры насчитывают тысячи пользователей, которые подключаются к серверу через Интернет. Несмотря на эти различия, разработка внеигровых ботов в обоих случаях пойдёт по одному и тому же плану. Прежде всего необходимо изучить протокол взаимодействия игрового клиента и сервера.

У приложения NetChess нет никакой защиты от реверс-инжиниринга и внеигровых ботов. Именно по этой причине нам так быстро удалось понять его протокол. Если вы попробуете проделать то же самое с современной онлайн-игрой, возникнут сложности. Скорее всего, вы не сможете так просто установить соответствие между действиями игрока и данными в перехваченных пакетах. Одни и те же действия могут менять байты по разным смещениям без какой-либо закономерности. Если вы столкнулись с подобным поведением, значит игра имеет систему защиты. Самый надёжный и распространённый подход для защиты трафика приложения – это шифрование.

В главе 3 мы применяли алгоритмы шифрования для защиты памяти приложения. Теперь рассмотрим, как с их помощью обезопасить сетевой трафик.

Криптосистема

Перед изучением практических примеров, рассмотрим понятие **криптосистемы**. Криптосистема – это набор криптографических алгоритмов для обеспечения конфиденциальности информации. Как правило, она предоставляет алгоритмы для следующих целей:

1. Генерация ключа.
2. Шифрование.
3. Дешифрование.

Первая категория алгоритмов в списке используется для создания **секретного ключа**, который удовлетворяет требованиям **шифра**.

Как работает шифрование? Предположим, что у нас есть некоторая информация (например сообщение), которое мы хотим защитить от несанкционированного чтения. Эта информация называется **открытый текст** (plaintext). Она вместе с секретным ключом передаётся алгоритму шифрования. После отработки алгоритм выдаст информацию в зашифрованном виде, который называется **шифротекст**. Чтобы снова получить открытый текст, необходимо передать шифротекст и ключ в алгоритм дешифрования. Это значит, что исходное сообщение смогут прочитать только те получатели, которые знают ключ.

Мы рассмотрели работу типичной криптосистемы в общих чертах. В реальных системах могут быть дополнительные шаги шифрования и дешифрования, а также возможности управления ключами.

Тестовое приложение

Для демонстрации алгоритмов шифрования воспользуемся простым приложением, которое передаёт текстовое сообщение по протоколу UDP. Мы использовали это приложение в разделе “Перехват трафика” (см. листинги 4-3 и 4-4). Немного изменим скрипт отправителя, чтобы вместо трёх байт отправлялась строка “Hello world!”.

Листинг 4-7. Скрипт `TestStringUdpSender.py`

```
import socket

def main():
  s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, 0)
  s.bind(("127.0.0.1", 24001))
  data = bytes("Hello world!", "utf-8")
  s.sendto(data, ("127.0.0.1", 24000))
  s.close()

```

```
10 if __name__
11   main()
```

Скрипт отправляет строку, хранящуюся в переменной `data`. Это байтовый массив, в котором каждой букве соответствует один байт (**кодировка ASCII**). Чтобы получить этот массив из исходной строки в кодировке **UTF-8**, используется функция `bytes`.

Запустите скрипт `TestUdpReceiver.py` из листинга 4-3 и `TestStringUdpSender.py`. Когда получатели примет сообщение, он выведет на консоль текст:

```
1 b'Hello world!'
```

Символ “b” в начале строки означает, что строка хранится в памяти в виде байтового массива.

Иллюстрация 4-22 демонстрирует перехваченный пакет тестового приложения.

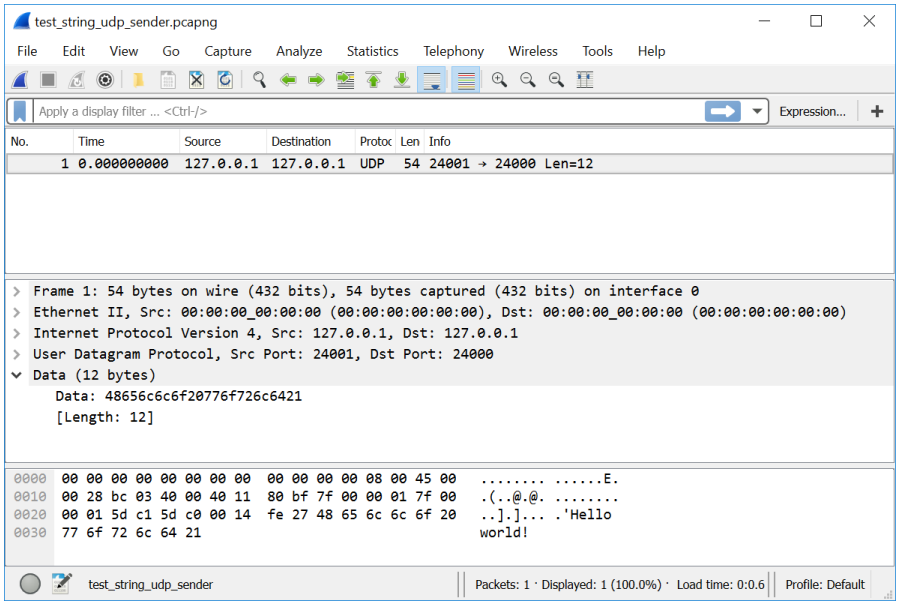


Иллюстрация 4-22. Перехваченный пакет тестового приложения

Wireshark корректно декодировал строку “Hello world!”. Мы можем её прочитать в нижней части окна анализатора в области байтового представления пакета.

Шифр XOR

Шифр XOR представляет собой одну из простейших криптосистем. Мы использовали его в главе 3 для сокрытия данных процесса от сканеров памяти. Теперь применим его для шифрования сетевого пакета.

Библиотека PyCrypto предоставляет реализацию шифра XOR. Мы воспользуемся ею вместо того, чтобы писать алгоритм самостоятельно.

В библиотеке PyCryptodome нет реализации шифра XOR. Если вы установили её, а не PyCrypto, вы не сможете запустить примеры из листингов 4-8, 4-9, 4-10 и 4-11.

Листинг 4-8 демонстрирует использование шифра XOR, предоставляемого библиотекой PyCrypto.

Листинг 4-8. Скрипт `XorTest.py`

```
from Crypto.Cipher import XOR

def main():
  key = b"The secret key"

  # Encryption
  encryption_suite = XOR.new(key)
  cipher_text = encryption_suite.encrypt(b"Hello world!")
  print(cipher_text)

  # Decryption
  decryption_suite = XOR.new(key)
  plain_text = decryption_suite.decrypt(cipher_text)
  print(plain_text)

if __name__ == '__main__':
  main()
```

Первая строка скрипта импортирует Python модуль `XOR`, в котором реализованы алгоритмы шифра. Чтобы ими воспользоваться, нам надо подготовить секретный ключ. Им служит строка “The secret key”, хранящаяся в переменной `key`.

Чтобы зашифровать строку, мы создаём объект `encryption_suite` класса `XORCipher` с помощью функции `new` (вызов `XOR.new`). В качестве параметра передаём в неё секретный ключ. У созданного объекта есть метод `encrypt`, который применяет шифр к переданному ему открытому тексту в формате

байтового массива. Получившийся шифротекст сохраняется в переменной `cipher_text` и выводится на консоль. Он выглядит следующим образом:

```
b'\x1c\r\tL\x1cE\x14\x1d\x17\x18DJ'
```

Оставшаяся часть функции `main` дешифрует шифротекст в исходный вид. Для этого мы создаём объект `dencryption_suite` точно так же, как и `encryption_suite` ранее. С помощью метода `decrypt` этого объекта мы дешифруем строку, хранящуюся в переменной `cipher_text`, и выводим результат на консоль. Он должен совпасть с исходной строкой “Hello world!”.

После внимательного изучения кода листинга 4-8 возникает вопрос. Можно ли использовать один и тот же объект класса `XORCipher` для шифрования и дешифрования? Ответ – нет. Классы библиотеки PyCrypto имеют внутреннее состояние, которое зависит от последней операции, выполненной с их помощью. Это означает, что любое действие над ними окажет влияние на последующее. Если вы зашифруете две строки друг за другом с помощью одного объекта, расшифровать их возможно только в той же последовательности. Иначе результат будет ошибочным. Надёжный и правильный способ использовать объекты `XORCipher` – использовать их для однократных операций шифрования и дешифрования.

Теперь применим шифр XOR для скриптов отправки и получения UDP-пакета нашего тестового приложения. Листинг 4-9 демонстрирует дополненный скрипт отправителя.

Листинг 4-9. Скрипт `XorUdpSender.py`

```
import socket
from Crypto.Cipher import XOR

def main():
  s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, 0)
  s.bind(("127.0.0.1", 24001))

  key = b"The secret key"
  encryption_suite = XOR.new(key)
  cipher_text = encryption_suite.encrypt(b"Hello world!")

  s.sendto(cipher_text, ("127.0.0.1", 24000))
  s.close()

if __name__ == '__main__':
  main()
```

В скрипте `XorUdpSender.py` мы шифруем строку “Hello world!” и отправляем её по протоколу UDP.

Скрипт получателя приведён в листинге 4-10.

Листинг 4-10. Скрипт `XorUdpReceiver.py`

```
import socket
from Crypto.Cipher import XOR

def main():
  s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, 0)
  s.bind(("127.0.0.1", 24000))
  data, addr = s.recvfrom(1024, socket.MSG_WAITALL)

  key = b"The secret key"
  decryption_suite = XOR.new(key)
  plain_text = decryption_suite.decrypt(data)
  print(plain_text)

  s.close()

if __name__ == '__main__':
  main()
```

Если вы запустите скрипты отправителя и получателя, результат будет тем же что и раньше. Скрипт `XorUdpReceiver.py` выведет на консоль полученную строку:

```
b'Hello world!'
```

Однако, если вы перехватите передаваемый пакет с помощью Wireshark, вы сразу заметите разницу. Этот пакет приведён на иллюстрации 4-23.

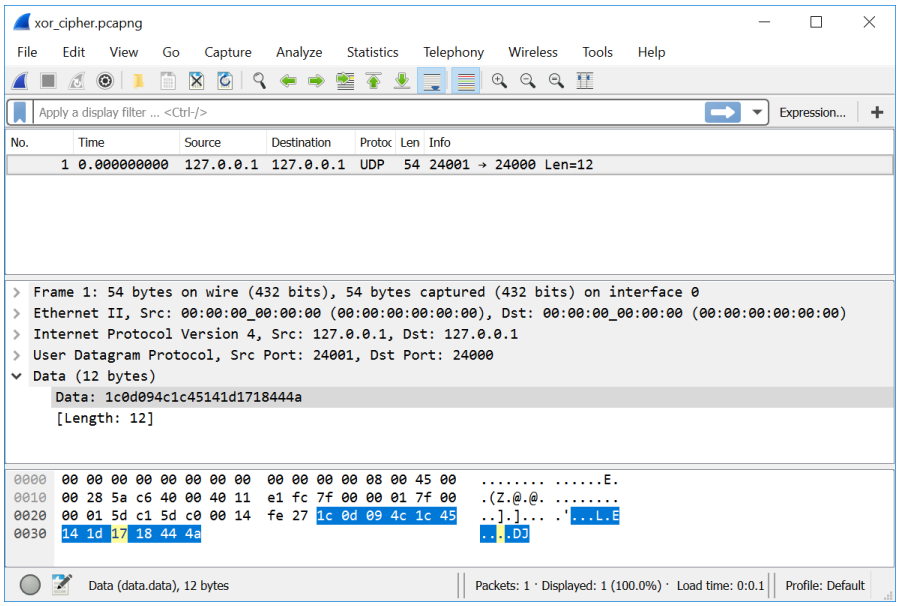


Иллюстрация 4-23. Перехваченный пакет, который был зашифрован XOR

Обратите внимание, что теперь Wireshark не может декодировать строку. Вы можете сделать это вручную, но только если вам известен секретный ключ.

Возможно, некоторые читатели решат, что шифр XOR – это отличный вариант для защиты приложения. Он прост в использовании и быстро работает. На самом деле его очень легко взломать. Рассмотрим подробнее, как это сделать.

В шифре применяется логическая операция **исключающее “или”**. Предположим, что мы шифруем открытый текст A с помощью секретного ключа K. Тогда получим шифротекст B:

```
1 A ⊕ K = B
```

Если мы применим исключающее “или” к A и B, то получим ключ K:

```
1 A ⊕ B = K
```

Это означает, что можно восстановить секретный ключ, если известны открытый текст и шифротекст. Скрипт `XorCrack.py` из листинга 4-11 восстанавливает ключ по рассмотренному алгоритму.

Листинг 4-11. Скрипт `XorCrack.py`

```
from Crypto.Cipher import XOR

def main():
  key = b"The secret key"

  # Encryption
  encryption_suite = XOR.new(key)
  cipher_text = encryption_suite.encrypt(b"Hello world!")
  print(cipher_text)

  # Decryption
  decryption_suite = XOR.new(key)
  plain_text = decryption_suite.decrypt(cipher_text)
  print(plain_text)

  # Crack
  crack_suite = XOR.new(plain_text)
  key = crack_suite.encrypt(cipher_text)
  print(key)

if __name__ == '__main__':
  main()
```

При запуске этот скрипт выведет на консоль следующее:

```
b'\x1c\r\tL\x1cE\x14\x1d\x17\x18DJ'
b'Hello world!'
b'The secret k'
```

Первая строка соответствует шифротексту. Далее идёт открытый текст и восстановленный секретный ключ.

Почему скрипт `XorCrack.py` восстановил только часть секретного ключа? В XOR шифре оператор исключающего “или” последовательно применяется к каждой букве открытого текста и соответствующему ей байту ключа. Если ключ оказался короче текста, оставшаяся его часть не используется. В противном случае он будет применяться циклично.

Как рассмотренное свойство оператора исключающего “или” поможет нам расшифровать пакет с реальными игровыми данными? В этом случае у нас есть только шифротекст. Наша задача – получить из него открытый текст. Прежде всего, необходимо восстановить секретный ключ. Для этого возьмём известный нам открытый текст и зашифруем его в точности той же криптосистемой, которой пользуется игровое приложение. Получив шифротекст, мы применим операцию исключающего “или” к нему и открытому тексту. Так мы узнаем ключ.

Например, мы заполняем форму регистрации для онлайн-игры. В ней надо указать информацию о новом игроке (имя, пароль, адрес электронной почты). Все эти данные нам известны. После заполнения формы, обычно требуется нажать кнопку “отправить”. Перехватим пакеты, которые игровое приложение посылает по этому нажатию. В них передаются данные пользователя из формы регистрации. Применим оператор исключающего “или” к введённой нами информации об игроке и шифротексту из пакета. Чтобы перепробовать все комбинации, понадобится время, но рано или поздно мы восстановим секретный ключ.

Можно заключить, что у шифра XOR есть положительные стороны, но он не способен обеспечить надёжную защиту для трафика приложения.

Шифр Triple DES

Следующий шифр, который мы рассмотрим, называется **Triple DES** (3DES). Для шифрования в нём троекратно применяется алгоритм **DES** (Data Encryption Standard), который был разработан в 1975 году компанией IBM. Сегодня DES считается ненадёжным из-за использования коротких секретных ключей длиной 56 бит. Современные компьютеры позволяют перебрать все возможные ключи такой длины (количеством 2^{56}) в течение нескольких дней. Алгоритм 3DES решает эту проблему путём увеличения длины ключа в три раза до 168 бит.

Почему необходимо применять алгоритм DES именно три раза? Разве не хватит двух? В этом случае мы получили бы ключ длиной 112 бит, которого достаточно для современных требований надёжности. Ожидается, что для взлома шифра потребуется перебрать 2^{112} всех возможных комбинаций. К сожалению, это предположение неверно. Атака под названием **встреча посередине** (meet-in-the-middle) позволяет сократить число вариантов ключей для перебора до 2^{57}. Этого недостаточно для надёжного шифрования открытого текста. Если же применить алгоритм 3DES, **атакующему** (лицу взламывающему шифр) придётся перебрать 2^{112} комбинаций ключей, даже если он применит атаку встреча посередине.

При разработке шифра 3DES учитывалось, насколько удобно будет его применение на специальных чипах. Сегодня по-прежнему эксплуатируется много устройств, выпущенных десять и более лет назад.

Они поддерживают алгоритм DES на аппаратном уровне. Эти устройства достаточно просто настроить на работу с шифром 3DES. Обратная совместимость с устаревшими решениями – это основная причина использования 3DES в наши дни. Более современные шифры быстрее и надёжнее.

Обе библиотеки PyCrypto и PyCryptodome предоставляют реализации шифров DES и 3DES. Мы рассмотрим только 3DES алгоритм.

Листинг 4-12 демонстрирует скрипт для шифрования и дешифрования строки с помощью 3DES.

Листинг 4-12. Скрипт `3DesTest.py`

```
from Crypto.Cipher import DES3
from Crypto import Random

def main():
  key = b"The secret key a"
  iv = Random.new().read(DES3.block_size)

  # Encryption
  encryption_suite = DES3.new(key, DES3.MODE_CBC, iv)
  cipher_text = encryption_suite.encrypt(b"Hello world!    ")
  print(cipher_text)

  # Decryption
  decryption_suite = DES3.new(key, DES3.MODE_CBC, iv)
  plain_text = decryption_suite.decrypt(cipher_text)
  print(plain_text)

if __name__ == '__main__':
  main()
```

В этом скрипте мы импортируем Python модули `DES3` и `Random` библиотеки PyCrypto. Первый из них предоставляет класс `DES3Cipher`, в котором реализованы алгоритмы шифрования и дешифрования. Модуль `Random` предоставляет генератор случайных последовательностей байтов. Его следует использовать вместо стандартного модуля `random`, распространяемого с интерпретатором Python. Потому что `random` считается небезопасным для целей шифрования.

Зачем алгоритму 3DES понадобился массив случайных байтов? 3DES – это блочный шифр. В нём открытый текст разделяется на блоки, которые последовательно шифруются с помощью секретного ключа. Если мы применим алгоритм как есть, шифр будет недостаточно надёжным. Причина в том, что атакующий может найти закономерность между отдельными блоками открытого текста и шифротекста. Тогда он сможет

определить или по крайней мере предположить содержимое зашифрованных блоков. Чтобы предотвратить эту уязвимость, надо смешать каждый блок открытого текста с предыдущим блоком шифротекста. Этот подход известен как **сцепление блоков шифротекста** (Cipher Block Chaining или CBC). Единственная проблема возникает с первым блоком открытого текста. С какими данными следует смешивать его? Решение заключается в использовании случайно сгенерированный данных. Они называются **вектором инициализации** (Initialization Vector или IV).

В скрипте `3DesTest.py` мы создаём **файлоподобный объект** (file-like) с помощью функции `new` модуля `Random`. После этого вызываем метод `read`, который возвращает массив случайных байтов указанной длины. Она должна быть равна длине одного блока, на которые разбивается открытый текст в алгоритме 3DES. В нашем случае это константа реализации `DES3.block_size`, равная восьми байтам. Мы сохраняем массив случайных байтов в переменной `iv`. Он будет смешиваться с первым блоком открытого текста при шифровании.

Возможно, вы заметили, что мы расширили секретный ключ двумя дополнительными символами до 16 байт. При использовании алгоритма 3DES длина ключа может быть либо 16, либо 24 байта.

После подготовки вектора инициализации и ключа, мы создаём объект `encryption_suite` класса `DES3Cipher` с помощью функции `new` модуля `DES3`. Она принимает три входных параметра:

1. Секретный ключ.
2. Режим сцепления блоков шифротекста.
3. Вектор инициализации (если он нужен для выбранного режима).

В скрипте `3DesTest.py` используется режим `DES3.MODE_CBC`. Библиотека PyCrypto предоставляет несколько альтернативных вариантов. Вы можете выбрать один из них.

При конструировании объектов для шифрования и дешифрования необходимо указывать один и тот же режим сцепления блоков шифротекста и вектор инициализации.

Интерфейс методов `encrypt` и `decrypt` класса `DES3Cipher` такой же, как и у `XORCipher`. Первый принимает на вход открытый текст, а второй – шифротекст.

После запуска скрипта `3DesTest.py`, вывод на консоли должен выглядеть следующим образом:

```
1 b'\xdc\xce\xf1^_\x95[\x16K\x93\x9a\xb8\x01\xf3\x1b\xcb'
2 b'Hello world!    '
```

Обратите внимание, что мы добавили четыре пробела в конце строки открытого текста “Hello world!”. Они необходимы, поскольку его длина должна быть кратна восьми байтам, т.е. длине блока шифрования. Это требование выбранного нами режима сцепления блоков шифротекста.

Теперь дополним скрипты отправки и приёма UDP сообщения так, чтобы они использовали 3DES шифр. Листинг 4-13 демонстрирует код отправителя.

Листинг 4-13. Скрипт `3DesUdpSender.py`

```
 1 import socket
 2 from Crypto.Cipher import DES3
 3 from Crypto import Random
 4
 5 def main():
 6   s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, 0)
 7   s.bind(("127.0.0.1", 24001))
 8
 9   key = b"The secret key a"
10   iv = Random.new().read(DES3.block_size)
11   encryption_suite = DES3.new(key, DES3.MODE_CBC, iv)
12   cipher_text = iv + encryption_suite.encrypt(b"Hello world!    ")
13
14   s.sendto(cipher_text, ("127.0.0.1", 24000))
15   s.close()
16
17 if __name__ == '__main__':
18   main()
```

Скрипт `3DesUdpSender.py` шифрует открытый текст так же, как и `3DesTest.py`. Единственное отличие в том, что мы добавляем вектор

инициализации в начало шифротекста. Затем отправляем его получателю в UDP-пакете. Для чего это нужно? Как вы помните, для дешифровки сообщения нужен секретный ключ и вектор инициализации. Ключ мы можем сгенерировать заранее и сохранить на стороне отправителя и получателя. К сожалению, проделать то же самое с вектором инициализации не получится. Он должен быть уникальным для каждой операции шифрования, иначе алгоритм будет скомпрометирован и атакующему будет проще взломать шифр. Следовательно, получатель сообщения должен каким-то образом узнать IV. Самое простое решение – отправлять его вместе с шифротекстом в одном пакете.

Возникает вопрос: безопасно ли передавать вектор инициализации в открытом виде? Да, это вполне безопасно. Главная задача IV – добавлять случайность в шифротекст. Благодаря ему мы получаем разный результат при шифровании одного и того же открытого текста. При применении криптосистем IV часто рассматривается как обязательная часть шифротекста.

Листинг 4-14 демонстрирует реализацию скрипта получателя.

Листинг 4-14. Скрипт `3DesUdpReceiver.py`

```
import socket
from Crypto.Cipher import DES3

def main():
  s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, 0)
  s.bind(("127.0.0.1", 24000))
  data, addr = s.recvfrom(1024, socket.MSG_WAITALL)

  key = b"The secret key a"
  decryption_suite = DES3.new(key, DES3.MODE_CBC, data[0:DES3.block_size])
  plain_text = decryption_suite.decrypt(data[DES3.block_size:])
  print(plain_text)

  s.close()

if __name__ == '__main__':
  main()
```

В скрипте `3DesUdpReceiver.py` мы передаём первый блок данных (байты с нулевого по `DES3.block_size`) из принятого UDP-пакета в функцию `new` в качестве вектора инициализации. Она конструирует объект `decryption_suite`, с помощью которого мы расшифровываем оставшиеся байты сообщения.

Если вы запустите сначала скрипт `3DesUdpReceiver.py`, а потом `3DesUdpSender.py`, получатель корректно расшифрует переданное сообщение и выведет его на консоль.

Вы можете использовать шифр 3DES в своих приложениях только тогда, когда на это есть серьёзные причины (например аппаратная поддержка со стороны используемого оборудования). Сегодня он не считается достаточно надёжным. Теоретические варианты атаки на шифр рассмотрены в этой статье. Кроме того, современные шифры работают быстрее 3DES.

Шифр AES

В 1998 году два бельгийских криптографа Винсент Рэймен и Йоан Даймен создали **шифр AES** (Advanced Encryption Standard). Он заменил DES и его вариации в качестве криптографического стандарта США.

В AES были решены проблемы шифра DES. Прежде всего он позволяет использовать длинные секретные ключи: 128, 192 и 256 бит. Любой из вариантов не вызовет накладных расходов алгоритма шифрования, как в случае 3DES. Их отсутствие – одна из причин высокой скорости работы AES. Возможность выбора появилась потому, что в AES длины блоков и ключа могут различаться.

Обе библиотеки PyCrypto и PyCryptodome предоставляют шифр AES. Интерфейс для его использования похож на 3DES.

Листинг 4-15 демонстрирует применение AES для шифрования и дешифрования строки.

Листинг 4-15. Скрипт `AesTest.py`

```
from Crypto.Cipher import AES
from Crypto import Random

def main():
  key = b"The secret key a"
  iv = Random.new().read(AES.block_size)

  # Encryption
  encryption_suite = AES.new(key, AES.MODE_CBC, iv)
  cipher_text = encryption_suite.encrypt(b"Hello world!    ")
  print(cipher_text)

  # Decryption
  decryption_suite = AES.new(key, AES.MODE_CBC, iv)
  plain_text = decryption_suite.decrypt(cipher_text)
  print(plain_text)
```

```
17
18 if __name__ == '__main__':
19   main()
```

Сравните скрипты `AesTest.py` и `3DesTest.py`. Они очень похожи. Функция `new` модуля `AES` создаёт объект `encryption_suite` класса `AESCipher`. У неё те же три входных параметра, что и в случае 3DES: секретный ключ, режим сцепления блоков, IV. Кроме того, AES поддерживает те же режимы сцепления, что и 3DES.

После запуска скрипта `AesTest.py`, в консоли напечатаются следующие строки:

```
1 b'\xed\xd5\x19]\x04\xba\xc5\x05^s\x18t\xa3\xb59x'
2 b'Hello world!    '
```

Нам опять пришлось дополнить открытый текст пробелами, до длины кратной восьми байтов. Это требование режима сцепления блоков `AES.MODE_CBC`.

Листинг 4-16 демонстрирует скрипт `AesUdpSender.py`, который шифрует сообщение алгоритмом AES и отправляет его.

Листинг 4-16. Скрипт `AesUdpSender.py`

```
1 import socket
2 from Crypto.Cipher import AES
3 from Crypto import Random
4
5 def main():
6   s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, 0)
7   s.bind(("127.0.0.1", 24001))
8
9   key = b"The secret key a"
10  iv = Random.new().read(AES.block_size)
11  encryption_suite = AES.new(key, AES.MODE_CBC, iv)
12  cipher_text = iv + encryption_suite.encrypt(b"Hello world!    ")
13
14  s.sendto(cipher_text, ("127.0.0.1", 24000))
15  s.close()
16
17 if __name__ == '__main__':
18   main()
```

Здесь мы отправляем IV в начале данных пакета точно так же, как и в скрипте `3DesUdpSender.py` (листинг 4-13). Алгоритм шифрования и отправки пакета такой же, как при использовании 3DES.

Скрипт `AesUdpReceiver.py` из листинга 4-17 получает и дешифрует сообщение.

Листинг 4-17. Скрипт `AesUdpReceiver.py`

```
import socket
from Crypto.Cipher import AES

def main():
  s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, 0)
  s.bind(("127.0.0.1", 24000))
  data, addr = s.recvfrom(1024, socket.MSG_WAITALL)

  key = b"The secret key a"
  decryption_suite = AES.new(key, AES.MODE_CBC, data[0:AES.block_size])
  plain_text = decryption_suite.decrypt(data[AES.block_size:])
  print(plain_text)

  s.close()

if __name__ == '__main__':
  main()
```

Скрипт `AesUdpReceiver.py` работает по тому же алгоритму, что и `3DesUdpReceiver.py` из листинга 4-14.

Попробуйте запустить скрипты отправителя и получателя, чтобы проверить корректность их работы.

Если вы собираетесь использовать **симметричный шифр** в своём приложении, всегда выбирайте AES вместо 3DES.

Предположим, что игровое приложение, для которого мы собираемся написать бота, применяет симметричный шифр для защиты своего сетевого трафика. Можем ли мы его взломать, чтобы изучить протокол игры? Если используются надёжные алгоритмы вроде 3DES или AES, скорее всего, придётся перебрать и проверить все возможные комбинации секретных ключей. Этот подход известен как **метод “грубой силы”**. Однако, существуют атаки на шифр, позволяющие уменьшить число ключей для перебора и проверки. Они специфичны для алгоритма шифрования, режима его работы, деталей реализации и качества выбранного секретного ключа.

Возникает другой вопрос. Если мы применили какую-то технику и получили набор ключей для перебора и проверки, как понять, что один из них подошёл? Ведь в большинстве случаев мы не знаем точно, как выглядит открытый текст.

Первое решение этого затруднения заключается в том, чтобы собрать информацию об открытом тексте. Мы можем прочитать состояния игровых объектов в окне приложения или проанализировать память его процесса. Высока вероятность, что эти состояния окажутся в одном из пакетов, которыми обменивается игровой клиент и сервер. Следовательно, если секретный ключ подойдёт, мы должны прочитать эти данные.

Альтернативное решение заключается в применении статистического теста к расшифрованным данным. Если проверяемый секретный ключ корректен, они должны быть более упорядоченными. Иначе мы получим набор случайных байтов без какой-либо закономерности.

Шифр RSA

Все рассмотренные нами ранее шифры (XOR, 3DES, AES) являются симметричными. Это означает, что для шифрования и дешифрования используется один и тот же секретный ключ. Следовательно, он должен быть у отправителя и получателя сообщения. Этот факт может навести на мысль: зачем вообще нужно взламывать надёжный шифр? Ведь по сути секретный ключ находится на стороне пользователя в памяти игрового клиента. Достаточно найти его и импортировать в код внеигрового бота. После этого он сможет взаимодействовать с сервером точно так же, как и оригинальный клиент.

Возникает встречный вопрос: возможно ли защитить секретный ключ на стороне игрового клиента? Лучшим решением было бы вообще не хранить его локально у пользователя. С другой стороны, сервер не может просто отправлять ключ перед началом каждого сеанса обмена пакетами. Если атакующий перехватит его, он легко расшифрует весь дальнейший трафик. **Асимметричное шифрование** решает именно эту проблему. Оно предоставляет алгоритмы для безопасной передачи ключа.

Рассмотрим асимметричный **шифр RSA**. Его идея заключается в том, чтобы применять **одностороннюю математическую функцию** для шифрования открытого текста. Ключ является входным параметром этой функции. Чтобы взломать шифр, необходимо решить математическое уравнение, то есть найти открытый текст по известному шифротексту и ключу. Однако, главная особенность односторонних функций заключается в сложности нахождения входного параметра по её известному значению. Поэтому взломать шифр за разумное время невозможно.

Если вычислить функцию **обратную** односторонней нельзя, как же тогда происходит дешифрование сообщения? Предположим, что мы зашифровали сообщение с помощью ключа и односторонней функции. Шифротекст передали получателю. Даже зная ключ, используемый при шифровании, он не сможет дешифровать сообщение. Нюанс заключается в том, что для асимметричного шифрования выбираются особенные односторонние функции: те у которых есть **лазейка**. Лазейка – эта некоторая подсказка, помогающая вычислить обратную функцию, т.е. получить открытый текст по известному шифротексту и ключу. Мы пришли к концепции двух ключей: первый для выполнения шифрования (известен как **открытый ключ**) и второй – лазейка для вычисления обратной функции (**закрытый ключ**).

Рассмотрим, как работает асимметричное шифрование с точки зрения пользователя. Наша задача – получить от другого лица зашифрованное сообщение. Сначала надо вычислить пару ключей: открытый и закрытый. Первый из них передаём отправителю информации. Он шифрует своё сообщение этим ключом и отправляет нам шифротекст. Благодаря закрытому ключу, который служит лазейкой к односторонней функции, мы расшифровываем сообщение. Как видно из рассмотренной схемы, атакующий может перехватить открытый ключ и шифротекст, но это не поможет ему расшифровать сообщение. Для этого нужен закрытый ключ, но его получатель хранит у себя и никому не передаёт.

Обе библиотеки PyCrypto и PyCryptodome предоставляют реализацию шифра RSA. Но в PyCryptodome отсутствуют некоторые недостаточно надёжные функции RSA.

Листинг 4-18 демонстрирует использование RSA для шифрования и дешифрования строки.

Листинг 4-18. Скрипт `RsaTest.py`

```
from Crypto.PublicKey import RSA
from Crypto import Random

def main():
  key = RSA.generate(1024, Random.new().read)

  # Encryption
  cipher_text = key.encrypt(b"Hello world!", 32)
  print(cipher_text)

  # Decryption
  plain_text = key.decrypt(cipher_text)
  print(plain_text)
```

```
14
15 if __name__ == '__main__':
16   main()
```

Скрипт `RsaTest.py` не заработает, если вы используете библиотеку PyCryptodome.

В скрипте мы импортируем два модуля Python `Random` и `RSA`. Первый из них нам уже известен. Второй предоставляет функции для генерации и применения открытого и закрытого ключа.

Сначала мы создаём объект `key` класса `_RSAobj` с помощью функции `generate` модуля `RSA`. Он содержит пару ключей (открытый и закрытый). Первый параметр функции обязательный. Он задаёт длину ключей (в нашем случае 1024 бита). Второй параметр опциональный. В нём передаётся функция генерации случайных чисел.

После создания объекта `key` мы вызываем его методы `encrypt` и `decrypt` для шифрования и дешифрования текста.

Может возникнуть вопрос: где применяются открытый и закрытый ключи в нашем примере? Ведь явно они нигде в коде не упоминаются. На самом деле шифрование и дешифрование происходит в одном и том же процессе, поэтому нет необходимости в передаче открытого ключа. Если же передача нужна, объект `key` предоставляет методы для экспорта и импорта ключей.

В листинге 4-18 мы рассмотрели использование шифра RSA самого по себе. В таком виде он уязвим для **атаки на основе подобранного открытого текста** (chosen-plaintext attack или CPA). Поэтому RSA всегда используют в комбинации со **схемой дополнения OAEP** (Optimal Asymmetric Encryption Padding), которая предотвращает эту уязвимость. Такая комбинация шифра и схемы дополнения известна как RSA-OAEP.

Листинг 4-19 демонстрирует использование RSA-OAEP алгоритма для шифрования строки.

Листинг 4-19. Скрипт `RsaOaepTest.py`

```
1 from Crypto.PublicKey import RSA
2 from Crypto.Cipher import PKCS1_OAEP
```

```
3 from Crypto import Random
4
5 def main():
6   key = RSA.generate(1024, Random.new().read)
7
8   # Encryption
9   encryption_suite = PKCS1_OAEP.new(key)
10  cipher_text = encryption_suite.encrypt(b"Hello world!")
11  print(cipher_text)
12
13  # Decryption
14  decryption_suite = PKCS1_OAEP.new(key)
15  plain_text = decryption_suite.decrypt(cipher_text)
16  print(plain_text)
17
18 if __name__ == '__main__':
19   main()
```

Теперь для шифрования и дешифрования мы используем не `key`, а объекты класса `PKCS10AEP_Cipher` из модуля `PKCS1_OAEP`. Он конструируется функцией `new`, которая принимает входным параметром объект класса `_RSAobj` (то есть ключи RSA). Для шифрования и дешифрования используются два разных OAEP-объекта: `encryption_suite` и `decryption_suite`.

Применим RSA-OAEP шифр для нашего тестового приложения, отправляющего UDP-пакет по сети. Прежде всего необходимо изменить его алгоритм. В случае симметричного шифрования он тривиален: зашифровать открытый текст, передать его в пакете, расшифровать на стороне получателя. При применении асимметричного шифра появляется дополнительный шаг: передача открытого ключа отправителю сообщения. Ведь с его помощью и будет происходить шифрование.

Рассмотрим пошагово новый алгоритм тестового приложения:

1. Скрипт отправителя сообщения запускается первым. Он создаёт UDP-сокет и ожидает получения открытого ключа.
2. Скрипт получателя запускается. Он создаёт UDP-сокет. Затем генерирует пару ключей.
3. Получатель сообщения посылает свой открытый ключ.
4. Отправитель читает ключ из пришедшего UDP-пакета и использует его для шифрования открытого текста по алгоритму RSA-OAEP.
5. Отправитель посылает шифротекст с сообщением.
6. Получатель принимает шифротекст и дешифрует его, используя свой закрытый ключ.

Листинг 4-20 демонстрирует скрипт, отправляющий сообщение.

Листинг 4-20. Скрипт `RsaUdpSender.py`

```
import socket
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

def main():
  s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, 0)
  s.bind(("127.0.0.1", 24001))

  public_key, addr = s.recvfrom(1024, socket.MSG_WAITALL)

  key = RSA.importKey(public_key)
  cipher = PKCS1_OAEP.new(key)

  cipher_text = cipher.encrypt(b"Hello world!")

  s.sendto(cipher_text, ("127.0.0.1", 24000))
  s.close()

if __name__ == '__main__':
  main()
```

В этом скрипте мы используем функцию `importKey` модуля `RSA`. Она конструирует объект класса `_RSAobj`, содержащий только открытый ключ. Этого объекта будет достаточно для шифрования, но не для дешифрования. На входе `importKey` принимает ключ в формате байтового массива, который мы получаем из UDP-пакета. Переменная `key` используется для конструирования объекта `cipher` класса `PKCS1OAEP_Cipher`. С его помощью мы шифруем сообщение и отправляем его получателю.

Скрипт, получающий сообщение, приведён в листинге 4-21.

Листинг 4-21. Скрипт `RsaUdpReceiver.py`

```
import socket
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto import Random

def main():
  s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, 0)
  s.bind(("127.0.0.1", 24000))

  key = RSA.generate(1024, Random.new().read)
  public_key = key.publickey().exportKey()

  s.sendto(public_key, ("127.0.0.1", 24001))

  data, addr = s.recvfrom(1024, socket.MSG_WAITALL)

  cipher = PKCS1_OAEP.new(key)
  plain_text = cipher.decrypt(data)
  print(plain_text)
```

```
20
21   s.close()
22
23 if __name__ == '__main__':
24   main()
```

Как мы рассмотрели ранее, в скрипте получателя сообщения появились дополнительные шаги для передачи открытого ключа. Мы генерируем пару ключей и сохраняем её в объекте `key`. Затем с помощью его метода `publickey` создаём временный объект класса `_RSAobj`, содержащий только открытый ключ. Его нужно представить в формате байтового массива, чтобы передать в UDP-пакете. Для этого вызываем метод `exportKey` временного объекта. Результат сохраняем в переменную `public_key`.

Метод `exportKey` есть у любого объекта класса `_RSAobj`. Что он экспортирует, если мы вызовем его для объекта `key`, содержащего и открытый ключ, и закрытый? В этом случае метод вернёт закрытый ключ. Это может быть полезно для сохранения его на жёстком диске и дальнейшего использования.

Открытый ключ мы посылаем в UDP-пакете без шифрования. Его перехват не поможет в атаке на шифр. После этого мы ждём пока отправитель получит ключ, зашифрует им сообщение и отправит его. Для дешифровки используется объект `cipher` класса `PKCS1OAEP_Cipher`, который применяет закрытый ключ в алгоритме RSA-OAEP.

Чтобы протестировать наше приложение, запустите сначала скрипт `RsaUdpSender.py`, а потом `RsaUdpReceiver.py`. Получатель должен вывести на консоль переданное сообщение.

По сравнению с симметричными шифрами RSA имеет один существенный недостаток – он работает значительно медленнее. Причина в том, что симметричные шифры используют в своих алгоритмах **операции битового сдвига** и логическое “или”. Современные процессоры обрабатывают их очень быстро за счёт специальных логических блоков, которые способны выполнять по одной операции за такт. Обычная **тактовая частота** сегодня составляет порядка 2.5 гигагерц (Гц). Это значит, что в секунду процессор способен совершать 2500000000 операций. Наличие нескольких ядер увеличивает это число.

Алгоритмы RSA используют математические функции: **возведение в степень по модулю** во время шифрования и вычисление **функции Эйлера** для дешифровки. Их расчёт не может быть ускорен с помощью специальных логических блоков, а потому требует большого числа тактов.

Проблема интенсивных вычислений в асимметричных шифрах решается с помощью **сеансового ключа**. Идея заключается в том, чтобы сгенерировать временный ключ для симметричного шифрования. В этом случае алгоритм RSA используется только для его безопасной передачи. После этого обе стороны переходят на симметричный шифр и с его помощью защищают свои сообщения. Временный ключ действует до окончания соединения. Для нового соединения он будет сгенерирован повторно.

Вы можете легко изменить скрипты `RsaUdpSender.py` и `RsaUdpReceiver.py` так, чтобы вместо строки “Hello world!” передавался сеансовый ключ (например, шифра AES). После этого скрипты смогут перейти на симметричное шифрование для дальнейшего обмена сообщениями.

Асимметричное шифрование позволяет устранить уязвимость, связанную с постоянным хранением секретного ключа на стороне игрового клиента.

Обнаружение внеигровых ботов

Мы рассмотрели криптографические алгоритмы для защиты трафика игрового приложения. Разработчик бота должен потратить достаточно времени на перехват сетевых пакетов и их дешифровку. Предположим, ему это удалось и он написал внеигрового бота для нашей игры. Что мы можем предпринять в этом случае?

На самом деле обнаружить внеигрового бота намного проще чем внутриигрового или кликера. Всё что нужно сделать – это реагировать на получение некорректных пакетов на стороне сервера.

Для примера рассмотрим простейший случай. Мы используем симметричное шифрование и постоянно храним секретный ключ на стороне игрового клиента. Бот импортирует этот ключ и использует его для взаимодействия с сервером. В этом случае обнаружить бота очень трудно. Но у любой онлайн-игры должен быть предусмотрен механизм обновления игрового клиента. Он необходим для исправления ошибок и

добавления новых возможностей. Одно из обновлений может менять секретный ключ без уведомления об этом пользователя. Очевидно, что на стороне сервера ключ также будет обновлён. Если после этого бот отправит пакет, зашифрованный старым ключом, сервер не сможет его корректно дешифровать. Таким образом бот себя обнаружит.

Разработчик бота может своевременно реагировать на обновления и импортировать новые ключи. Однако, мы обнаружим и заблокируем всех пользователей, которые используют старую версию бота. Обычно игроки покупают и запускают бота, не понимая основных принципов его работы. Поэтому очень часто они попадаются на использовании его старых версий.

В случае асимметричного шифрования, мы можем применить тот же подход для обнаружения бота. Есть несколько вариантов распределения ключей. Предположим, что сервер постоянно хранит у себя открытый ключ игрового клиента. В начале сеанса клиент посылает свой открытый ключ. Сервер сравнивает его со своей копией. Если обнаруживается различие, велика вероятность, что пользователь запустил бота. Если ключи совпали, сервер отправляет свой открытый ключ клиенту. После этого они могут шифровать сообщения друг для друга. При обновлении мы генерируем заново все ключи: пару открытый-закрытый на стороне клиента, только открытый ключ клиента на сервере. Если бот попробует воспользоваться старыми ключами, мы его обнаружим.

Если вы не хотите генерировать новые ключи шифрования, есть альтернативное решение. Вы можете регулярно менять протокол игрового приложения. Изменение может быть незначительным. Например, будет достаточно поменять порядок параметров игровых объектов в сетевом пакете или увеличить номер версии протокола. После этого проверив принятый от клиента пакет на соответствие новому формату, будет просто обнаружить бота.

Специальные техники

В этой главе мы рассмотрим специальные техники разработки игровых ботов. Они применяются в особых случаях для обхода некоторых видов защит от кликеров и внутриигровых ботов.

Сначала мы познакомимся с эмуляцией стандартных устройств ввода: клавиатуры и мыши. Затем перейдём к более сложной технике перехвата вызовов процесса игрового приложения к WinAPI библиотекам.

Эмуляция устройств ввода

Рассмотрим технику эмуляции устройств ввода. Этот подход применяется для обхода защит от кликеров, которые проверяют состояние клавиатуры. Алгоритм работы таких защит подробно разобран во второй главе.

Когда мы используем вместо клавиатуры или мыши эмулятор, у ОС нет возможности обнаружить подмену. Симулируемые эмулятором события (например, нажатия клавиш) будут обрабатываться ОС точно так же, как и для настоящей клавиатуры. Поэтому защите игрового приложения будет намного сложнее различать действия бота и игрока.

Инструменты для разработки

Прежде всего нам следует выбрать устройство, которое будет выполнять роль эмулятора. Рассмотрим основные требования к нему:

- Невысокая цена.
- Средства разработки (IDE и компилятор) должны быть бесплатны.
- Среда разработки должна предоставлять библиотеки для эмуляции устройств ввода.
- Должна быть доступная подробная документация.

Плата Arduino удовлетворяет всем перечисленным требованиям. Кроме того, Arduino — это одна из лучших аппаратных платформ, чтобы познакомиться с разработкой программ для **встраиваемых систем**.

Следующий вопрос, который следует решить: какую версию платы Arduino выбрать? Чтобы ответить на него, изучим возможности средств разработки. Arduino IDE предоставляет библиотеки для эмуляции клавиатуры и мыши. Согласно документации, некоторые версии плат их не поддерживают. Следовательно, нам они не подойдут. Нас устроят следующие модели: Leonardo, Micro и Due.

Мы выбрали аппаратную платформу. Теперь самое время установить средства разработки для неё. Компания производитель плат Arduino предоставляет бесплатную IDE с интегрированным C++ компилятором и библиотеками для поддержки периферии. Скачайте её с официального сайта и установите.

Теперь установим драйвер для работы с платой Arduino. Для этого нужна программа установки из каталога Arduino IDE. Её путь по умолчанию: `C:\Program Files (x86)\Arduino\drivers`. В каталоге `drivers` есть две программы: `dpinst-amd64.exe` для 64-разрядной версии Windows и `dpinst-x86.exe` для 32-разрядной. Выберите подходящую вам и перед её запуском подключите плату к компьютеру с помощью USB кабеля.

После установки драйвера выполните заключительные шаги конфигурации в Arduino IDE:

1. Прочитайте модель вашей платы. Для этого в главном меню выберите пункт “Tools” -> “Get Board Info” (“Инструменты” -> “Информация о плате”). Проверьте, что в пункте меню “Tools” -> “Board:…” (“Инструменты” -> “Плата:…”) модель указана правильно.
2. Укажите порт подключения платы в пункте главного меню “Tools”->”Port:…” (“Инструменты” -> “Порт:…”).

Теперь Arduino IDE настроена и готова к работе.

Самой по себе платы Arduino недостаточно для эмуляции устройств ввода. Мы должны написать для неё программу, которая посылала бы ОС события о симулируемых действиях. Со стороны компьютера этой программой будет управлять бот-кликер, написанный на языке AutoIt. Для такого взаимодействия понадобится набор AutoIt скриптов **CommAPI**.

Эмуляция клавиатуры

Есть два варианта реализации бота, использующего эмулятор устройства ввода.

В первом случае все алгоритмы бота реализованы в программе, работающей на плате Arduino. После её загрузки на устройство, всё готово к работе. Бот запускается автоматически, как только вы подключите плату к компьютеру через USB. Такая архитектура лучше всего подходит для “слепых” ботов, которые нажимают кнопки, не проверяя состояние игровых объектов. К сожалению, программа, запущенная на Arduino не имеет доступа к WinAPI-интерфейсу. Следовательно, она не сможет прочитать данные из процесса игрового приложения или устройства вывода.

Если ваш бот должен реагировать на игровые события, следует выбрать второй вариант реализации. В этом случае его алгоритмы запускаются и работают на компьютере. Программа платы Arduino отвечает только за симуляцию событий устройства ввода. В такой схеме бот имеет полный доступ к WinAPI и может читать состояние игровых объектов. После принятия решения, он отправляет плате Arduino команду на симуляцию нужного действия.

Мы рассмотрим пример второго варианта реализации бота. Он более надёжен и универсален.

Интерфейс взаимодействия платы и бота может быть любым: Ethernet, UART, I2C, SPI. Предлагаю остановиться на самом простом варианте, не требующем дополнительного оборудования кроме самой платы и USB провода. Речь идёт об интерфейсе **UART** (Universal Asynchronous Receiver-Transmitter).

Листинг 5-1 демонстрирует программу `keyboard.ino` для платы Arduino. Она симулирует события клавиатуры. При этом из UART-интерфейса читается код клавиши, которую требуется нажать.

Листинг 5-1. Программа `keyboard.ino`

```
#include <Keyboard.h>

void setup()
{
  Serial.begin(9600);
  Keyboard.begin();
}

void loop()
```

```
{
  if (Serial.available() > 0)
  {
    int incomingByte = Serial.read();
    Keyboard.write(incomingByte);
  }
}
```

В этой программе мы используем библиотеку **Keyboard**, которую предоставляет Arduino IDE. Она позволяет генерировать события нажатия клавиш. Подключённый по USB компьютер получает их через интерфейс **HID** (Human Interface Device). Он является современным стандартом взаимодействия с устройствами ввода.

Оба интерфейса HID и UART способны работать одновременно по одному USB-кабелю, соединяющему плату Arduino и компьютер.

В первой строке программы мы включаем заголовок `Keyboard.h`. В нём создаётся глобальный объект `Keyboard` класса `Keyboard_`. Все возможности библиотеки доступны через его методы.

В нашей программе всего две функции: `setup` и `loop`. Возможно, вы помните, что в любом C++ приложении обязательно должна быть ещё функция `main`. Она генерируется IDE во времени компиляции. В ней выполняется два действия: однократный вызов `setup` и цикличный вызов `loop`. **Прототипы** обеих этих функций предопределены, и поменять их нельзя.

Кроме `Keyboard` мы используем глобальный объект `Serial`. Он предоставляет доступ к интерфейсу UART. Для инициализации обоих объектов в функции `setup` вызываются методы `begin`. Для `Serial` этот метод принимает входным параметром скорость передачи данных между компьютером и платой, которая в нашем случае равна 9600 бит/с. У метода `begin` объекта `Keyboard` нет входных параметров. Сразу после его вызова плата начинает эмулировать клавиатуру.

После выполнения функции `setup` Arduino плата готова принимать команды по UART интерфейсу и симулировать нажатия соответствующих

клавиш. За это отвечает код функции `loop`. Её алгоритм состоит из трёх шагов:

1. С помощью метода `available` объекта `Serial` проверить, были ли получены данные по UART интерфейсу. Они сохраняются во входном буфере платы, размер которого 64 байта. Метод возвращает количество принятых байт. Если передачи не было, вернётся значение ноль.
2. Прочитать один байт из входного буфера UART с помощью метода `read` объекта `Serial`. Байт интерпретируется как ASCII код клавиши, нажатие которой следует симулировать.
3. Симулировать нажатие клавиши через HID-интерфейс с помощью метода `write` объекта `Keyboard`. Подключённый по USB компьютер обработает его как событие обычной клавиатуры.

Чтобы скомпилировать программу `keyboard.ino` и загрузить её на плату, откройте её в Arduino IDE и нажмите комбинацию клавиш Ctrl+U.

Мы подготовили плату. Теперь разработаем AutoIt скрипт, который будет ею управлять. Он должен посылать через UART интерфейс ASCII коды клавиш. Функции работы с UART предоставляет ОС через WinAPI. Доступ к ним из языка AutoIt могут значительно упростить обёртки CommAPI. Скачайте и скопируйте их в каталог вашего скрипта. Проверьте, что все необходимые файлы на месте:

- `CommAPI.au3`
- `CommAPIConstants.au3`
- `CommAPIHelper.au3`
- `CommInterface.au3`
- `CommUtilities.au3`

Листинг 5-2 демонстрирует использование обёрток CommAPI. Приведённый в нём скрипт печатает строку “Hello world!” в окне Notepad. Для симуляции нажатий клавиш он использует плату Arduino с загруженной на неё программой из листинга 5-1.

Листинг 5-2. Скрипт `ControlKeyboard.au3`

```
#include "CommInterface.au3"

func ShowError()
    MsgBox(16, "Error", "Error " & @error)
    endfunc
```

```
 6
 7 func OpenPort()
 8     local const $iPort = 7
 9     local const $iBaud = 9600
10     local const $iParity = 0
11     local const $iByteSize = 8
12     local const $iStopBits = 1
13
14     $hPort = _CommAPI_OpenCOMPort($iPort, $iBaud, $iParity, $iByteSize, $iStopBits)
15     if @error then
16         ShowError()
17         return NULL
18     endif
19
20     _CommAPI_ClearCommError($hPort)
21     if @error then
22         ShowError()
23         return NULL
24     endif
25
26     _CommAPI_PurgeComm($hPort)
27     if @error then
28         ShowError()
29         return NULL
30     endif
31
32     return $hPort
33 endfunc
34
35 func SendArduino($hPort, $command)
36     _CommAPI_TransmitString($hPort, $command)
37     if @error then ShowError()
38 endfunc
39
40 func ClosePort($hPort)
41     _CommAPI_ClosePort($hPort)
42     if @error then ShowError()
43 endfunc
44
45 $hWnd = WinGetHandle("[CLASS:Notepad]")
46 WinActivate($hWnd)
47 Sleep(200)
48
49 $hPort = OpenPort()
50
51 SendArduino($hPort, "Hello world!")
52
53 ClosePort($hPort)
```

Общий алгоритм скрипта состоит из следующих шагов:

1. Переключиться на окно Notepad с помощью AutoIt функции `WinActivate`.
2. Установить **последовательное соединение** (serial communication) по интерфейсу UART с платой Arduino, используя функцию `OpenPort`.
3. Отправить команду набора строки “Hello world!” на плату с помощью функции `SendArduino`.
4. Закрыть последовательное соединение функцией `ClosePort`.

Рассмотрим подробнее работу функций `OpenPort`, `SendArduino` и `ClosePort`.

Функция `OpenPort` устанавливает соединение и подготавливает плату Arduino к взаимодействию. Она возвращает дескриптор соединения. В ней происходят следующие вызовы CommAPI:

1. `_CommAPI_OpenCOMPort` устанавливает последовательное соединение с указанными параметрами. Из них `iParity`, `iByteSize` и `iStopBits` одинаковы для Arduino плат всех моделей. Параметр `iBaud` задаёт скорость передачи данных. Она должна соответствовать скорости, переданной в метод `begin` объекта `Serial` в программе платы. Параметр `iPort` определяет номер **последовательного порта** (COM порта), через который плата подключена к компьютеру. На самом деле подключение происходит по USB, а COM порт эмулируется. Уточнить номер порта можно в пункте меню “Tools” -> “Port:…” (“Инструменты” -> “Порт:…”) Arduino IDE. Например, если там указан COM7, параметр `iPort` должен быть равен 7.
2. `_CommAPI_ClearCommError` возвращает код ошибки при передаче данных. Через второй необязательный параметр функции возвращается текущее состояние подключённого устройства. В нашем случае он не используется. Функция вызывается для сброса флага ошибки на стороне платы. Это действие очень важно, поскольку передача данных будет заблокирована до тех пор, пока флаг ошибки взведён.
3. `_CommAPI_PurgeComm` отменяет все текущие операции по передаче данных, а также очищает входной и выходной буферы подключённого устройства. После завершения работы этой функции Arduino готова принимать команды по UART.

Функция `SendArduino` представляет собой обёртку над вызовом `_CommAPI_TransmitString`, который передаёт указанную строку по UART интерфейсу.

Функция `ClosePort` закрывает соединение по переданному в неё дескриптору.

Вспомогательная функция `ShowError` нужна для отладки. Она выводит сообщение с кодом ошибки, которая может произойти на любом этапе установки соединения.

Чтобы протестировать скрипт, выполните следующие действия:

1. Подключите Arduino плату с загруженной на неё программой `keyboard.ino` к компьютеру с помощью USB кабеля.
2. Запустите приложение Notepad.
3. Запустите скрипт `ControlKeyboard.au3`.

В результате в окне Notepad будет набран текст “Hello world!”.

Сочетание клавиш

Разработанная нами программа `keyboard.ino` успешно справляется с симуляцией нажатия одной клавиши за раз. Однако в некоторых играх может понадобится симулировать **сочетание клавиш**, например Ctrl+Z. В этом случае одного байта для передачи команды будет недостаточно. Кроме кода основной клавиши нужно отправлять код **клавиши-модификатора**. Таким образом, программа должна уметь читать два байта из входного буфера UART интерфейса.

Рассмотрим методы объекта `Serial`. Раньше мы использовали `read`, но с его помощью можно прочитать только один байт из входного буфера UART. Есть альтернативный метод `readBytes`, который читает последовательность байт указанной длины. Первым параметром в него передаётся массив, в который будут сохранены данные. Вторым – его размер. Метод возвращает количество прочитанных байтов. Оно может отличаться от значения второго параметра, если буфер содержит меньше данных.

Задумаемся над вопросом: достаточно ли будет передавать только коды модификатора и клавиши? На самом деле, если по какой-то причине приём данных на плате начнётся с середины команды, возникнут серьёзные сложности. Второй байт этой команды будет интерпретирован как первый. Первый же байт следующей команды – как второй. В результате будет симулировано нажатие не той клавиши, которую ожидает управляющий скрипт. Из-за возникшего сдвига все последующие команды также выполнятся неверно.

Возможна ли ситуация, когда плата получает очередную команду не с начала? Если мы подключаем устройство до запуска управляющего скрипта, это маловероятно. Однако такая ситуация возможна, если плата

перезагрузится например из-за отошедшего USB разъёма или ошибки драйвера Windows.

Проблему можно решить с помощью **преамбулы**. Преамбула – это предопределённое значение, которое сигнализирует о начале команды. Для неё мы выделим первый байт сообщения. Теперь мы легко отличим начало передачи. Если программа Arduino получила первый байт и он отличается от преамбулы, значит команда читается со сдвигом и её лучше проигнорировать.

По сути мы разработали простейший протокол для передачи команд эмулятору по UART интерфейсу. В таблице 5-1 приведены значения каждого байта в сообщении.

Таблица 5-1. Формат команды

Номер байта	Значение
1	Преамбула.
2	Код клавиши-модификатора.
3	Код основной клавиши.

Рассмотрим пример команды для симуляции нажатия Alt+Tab. В этом случае управляющий скрипт отправляет три байта:

```
0xDC 0x82 0xB3
```

Первый из них (0xDC) – это преамбула. Дальше идёт код клавиши-модификатора 0x82, который соответствует Alt. Последний байт 0xB3 – это код клавиши Tab.

Листинг 5-3 демонстрирует Arduino программу, поддерживающую наш протокол.

Листинг 5-3. Программа `keyboard-combo.ino`

```
#include <Keyboard.h>

void setup()
{
  Serial.begin(9600);
  Keyboard.begin();
}

void pressKey(char modifier, char key)
{
  Keyboard.press(modifier);
  Keyboard.write(key);
```

```
  Keyboard.release(modifier);
}

void loop()
{
  static const char PREAMBLE = 0xDC;
  static const uint8_t BUFFER_SIZE = 3;

  if (Serial.available() > 0)
  {
    char buffer[BUFFER_SIZE] = {0};
    uint8_t readBytes = Serial.readBytes(buffer, BUFFER_SIZE);

    if (readBytes != BUFFER_SIZE)
      return;

    if (buffer[0] != PREAMBLE)
      return;

     pressKey(buffer[1], buffer[2]);
  }
}
```

В программе появилась новая функция `pressKey`. Кроме этого, алгоритм `loop` стал сложнее. Мы читаем принятую команду из входного буфера UART с помощью метод `readBytes` объекта `Serial`. Для проверки её корректности используем операторы `if`. Первый из них сравнивает длину команды с ожидаемой. Второй — соответствие её первого байта и преамбулы. Если любая из проверок не проходит, обработка команды прекращается.

Симуляция нажатия сочетания клавиш происходит в функции `pressKey`. У неё два входных параметра: код модификатора и клавиши. Чтобы нажать и удерживать модификатор, используется метод `press` объекта `Keyboard`. Затем симулируется нажатие основной клавиши с помощью метода `write`. После этого модификатор отпускается вызовом `release`.

Управляющий AutoIt скрипт также должен поддерживать новый протокол передачи команд. Его исправленная версия приведена в листинге 5-4.

Листинг 5-4. Скрипт `ControlKeyboardCombo.au3`

```
#include "CommInterface.au3"

func ShowError()
    MsgBox(16, "Error", "Error " & @error)
endfunc

func OpenPort()
    local const $iPort = 7
    local const $iBaud = 9600
    local const $iParity = 0
    local const $iByteSize = 8
    local const $iStopBits = 1
```

```
13
14      $hPort = _CommAPI_OpenCOMPort($iPort, $iBaud, $iParity, $iByteSize, $iStopBits)
15      if @error then
16          ShowError()
17          return NULL
18      endif
19
20      _CommAPI_ClearCommError($hPort)
21      if @error then
22          ShowError()
23          return NULL
24      endif
25
26      _CommAPI_PurgeComm($hPort)
27      if @error then
28          ShowError()
29          return NULL
30      endif
31
32      return $hPort
33  endfunc
34
35  func SendArduino($hPort, $modifier, $key)
36      local $command[3] = [0xDC, $modifier, $key]
37
38      _CommAPI_TransmitString($hPort, _
39          StringFromASCIIArray($command, 0, UBound($command), 1))
40
41      if @error then ShowError()
42  endfunc
43
44  func ClosePort($hPort)
45      _CommAPI_ClosePort($hPort)
46      if @error then ShowError()
47  endfunc
48
49  $hWnd = WinGetHandle("[CLASS:Notepad]")
50  WinActivate($hWnd)
51  Sleep(200)
52
53  $hPort = OpenPort()
54
55  SendArduino($hPort, 0x82, 0xB3)
56
57  ClosePort($hPort)
```

Единственное отличие здесь от скрипта `ControlKeyboard.au3` в функции `SendArduino`. Теперь вместо строки символов, которые передаются последовательно, она передаёт команду из трёх байтов: преамбула, модификатор и клавиша. Для отправки данных используется та же CommAPI функция `_CommAPI_TransmitString`. Сложность заключается в том, что она ожидает входным параметром строку. Команда же представляет собой байтовый массив. Его можно преобразовать в строку с помощью стандартной функции AutoIt `StringFromASCIIArray`.

Для тестирования Arduino программы и скрипта выполните следующие шаги:

1. Загрузите программу `keyboard-combo.ino` на Arduino плату.
2. Откройте несколько окон на компьютере.
3. Запустите скрипт `ControlKeyboardCombo.au3`.

Скрипт будет симулировать нажатие сочетания клавиш Alt+Tab и переключаться между открытыми окнами.

Эмуляция мыши

С помощью платы Arduino можно эмулировать не только клавиатуру, но и мышь.

Все библиотеки Arduino IDE рассчитаны на разработку устройств на основе платы. Например, уже знакомая нам библиотека Keyboard. С её помощью мы могли бы собрать и запрограммировать свою собственную клавиатуру. Но вместо этого мы использовали её для эмуляции настоящего устройства. Keyboard отлично подошла для решения этой задачи.

У Arduino IDE есть библиотека **Mouse**. Она аналогична Keyboard, но служит для разработки сходных с мышью устройств (например трекболы или джойстики). Mouse хорошо справляется со своей основной целью, но для эмуляции мыши её использовать неудобно.

Проблема в том, что библиотека оперирует относительными координатами курсора. Чем продиктовано такое решение? Представьте, что вы разрабатываете свою мышь на основе платы Arduino. Её перемещения по столу читаются с помощью светодиода-сенсора. Этот сенсор может сообщить на сколько единиц расстояния произошёл сдвиг относительно прошлого положения устройства. Значение сдвига посылается на компьютер через HID интерфейс, и ОС отрисовывает курсор в новой позиции экрана. Абсолютные координаты в эту схему не укладываются, поскольку светодиод-сенсор не способен установить расположение мыши относительно какой-либо точки стола.

Для нашей цели эмуляции устройства абсолютные координаты были бы удобнее. По ним управляющий AutoIt скрипт читает пиксели экрана. Он знает, в какой именно точке нужно совершить щелчок мыши. Поэтому

было бы естественно для скрипта указывать именно абсолютные координаты экрана.

У этой проблемы есть два возможных решения:

1. На стороне управляющего скрипта – реализовать алгоритм для расчёта относительных координат целевой точки.
2. На стороне программы платы – исправить библиотеку Mouse так, чтобы она работала с абсолютными координатами.

Сообщество пользователей Arduino уже решило задачу модификации библиотеки Mouse. Необходимые для этого изменения описаны в статье. К сожалению, это решение подходит только для Arduino IDE старой версий 1.0. В ней библиотеки Keyboard и Mouse были объединены в одну под название HID.

Чтобы исправить библиотеку Mouse в новых версиях IDE, выполните следующие действия:

1. Скачайте файл `Mouse.cpp` из архива примеров к этой книге.
2. Скопируйте его с заменой в каталог Arduino IDE. Путь по умолчанию должен быть `C:\Program Files (x86)\Arduino\libraries\Mouse\src`.

Также вы можете исправить файл `Mouse.cpp` самостоятельно. Для этого объявите макрос `ABSOLUTE_MOUSE_MODE` и измените часть массива `_hidReportDescriptor` следующим образом:

```
#define ABSOLUTE_MOUSE_MODE

static const uint8_t _hidReportDescriptor[] PROGMEM = {
...
#ifdef ABSOLUTE_MOUSE_MODE
    0x15, 0x01,                    //    LOGICAL_MINIMUM (1)
    0x25, 0x7F,                    //    LOGICAL_MAXIMUM (127)
    0x75, 0x08,                    //    REPORT_SIZE (8)
    0x95, 0x03,                    //    REPORT_COUNT (3)
    0x81, 0x02,                    //    INPUT (Data,Var,Abs)
#else
    0x15, 0x81,                    //    LOGICAL_MINIMUM (-127)
    0x25, 0x7f,                    //    LOGICAL_MAXIMUM (127)
    0x75, 0x08,                    //    REPORT_SIZE (8)
    0x95, 0x03,                    //    REPORT_COUNT (3)
    0x81, 0x06,                    //    INPUT (Data,Var,Rel)
#endif
```

В массиве `_hidReportDescriptor` перечислены данные, которые плата может отправить и получить от компьютера. Другими словами в нём описан протокол передачи данных. Благодаря ему компьютер может взаимодействовать со всем HID устройствами единообразно.

Если макрос `ABSOLUTE_MOUSE_MODE` объявлен, протокол будет изменён в двух местах:

1. Значение байта `LOGICAL_MINIMUM` с ID равным 0x15 изменено с -127 (0x81 в шестнадцатеричной системе) на 1. Таким образом мы задали минимально допустимое значение координаты курсора. Для относительной координаты оно может быть отрицательным, но не абсолютной.
2. Значение байта `INPUT` с ID равным 0x81 изменено с 0x06 на 0x02. Это означает, что теперь будут передаваться абсолютные координаты, а не относительные.

Чтобы переключиться обратно в режим относительных координат, просто удалите или закомментируйте объявление макроса `ABSOLUTE_MOUSE_MODE`:

```
1 #define ABSOLUTE_MOUSE_MODE
```

Программа `mouse.ino` из листинга 5-5 симулирует нажатие кнопки мыши в указанной точке экрана.

Листинг 5-5. Программа `mouse.ino`

```
 1 #include <Mouse.h>
 2
 3 void setup()
 4 {
 5   Serial.begin(9600);
 6   Mouse.begin();
 7 }
 8
 9 void click(signed char x, signed char y, char button)
10 {
11   Mouse.move(x, y);
12   Mouse.click(button);
13 }
14
15 void loop()
16 {
17   static const char PREAMBLE = 0xDC;
18   static const uint8_t BUFFER_SIZE = 4;
19
20   if (Serial.available() > 0)
21   {
22     char buffer[BUFFER_SIZE] = {0};
23     uint8_t readBytes = Serial.readBytes(buffer, BUFFER_SIZE);
24
```

```
25      if (readBytes != BUFFER_SIZE)
26        return;
27
28      if (buffer[0] != PREAMBLE)
29        return;
30
31     click(buffer[1], buffer[2], buffer[3]);
32    }
33 }
```

Алгоритмы программ `mouse.ino` и `keyboard-combo.ino` из листинга 5-3 очень похожи. Теперь мы получаем от управляющего AutoIt скрипта команду, состоящую не из трёх байт, а из четырёх. Её формат приведён в таблице 5-2.

Таблица 5-2. Формат команды

Номер байта	Значение
1	Преамбула.
2	Координата X-точки, в которой следует симулировать нажатие кнопки.
3	Координата Y-точки.
4	Код кнопки мыши, которая будет нажата.

Получив команду по UART интерфейсу, мы проверяем её длину и корректность первого байта преамбулы. Если оба условия выполнены, вызываем функцию `click`. Для симуляции действий мыши используется глобальный объект `Mouse`. Он инициализируется с помощью метода `begin` точно так же, как и `Keyboard`. Перед тем как нажать кнопку, необходимо переместить курсор в заданную координату. Для этого вызываем метод `move` объекта `Mouse`, в который передаём координаты X и Y целевой точки. Затем с помощью метода `click` симулируем нажатие в текущей позиции курсора.

Внимательный читатель заметит, что максимально допустимые значения координат X и Y ограничены числом 127. В шестнадцатеричном виде оно равно 0x7F. Это максимальное [целое положительное число](#) со знаком, которое может быть передано в одном байте. Это ограничение продиктовано протоколом HID. Обратите внимание на значение байта `LOGICAL_MAXIMUM` в массиве `_hidReportDescriptor`:

```
1     0x25, 0x7f,                    //     LOGICAL_MAXIMUM (127)
```

Получается, что максимальные координаты, на которые может переместить курсор Arduino плата, равны 127×127. Однако разрешение современных мониторов значительно превышает эти числа. Перекладка координат HID устройства в координаты монитора происходит на уровне ОС. Придётся повторить её в нашем управляющем AutoIt скрипте, чтобы правильно спозиционировать курсор.

Итак, скрипт знает абсолютные координаты точки экрана, в которой следует симулировать нажатие кнопки мыши. Задача заключается в том, чтобы перевести эти координаты в шкалу Arduino платы.

Формулы перевода координат выглядят следующим образом:

```
1 Xa = 127 * X / Xres
2 Ya = 127 * Y / Yres
```

Значения переменных приведены в таблице 5-3.

Таблица 5-3. Переменные в формулах перевода координат

Переменные	Значение
Xa, Ya	Координаты X и Y в шкале Arduino.
X, Y	Координаты X и Y в шкале экрана.
Xres, Yres	Разрешение экрана в пикселях.

Рассмотрим пример перевода координат с помощью формул. Предположим, что разрешение нашего экрана 1366×768. Управляющий скрипт симулирует нажатие кнопки мыши в точке с координатами экрана X = 250 и Y = 300. Тогда ему надо отправить плате Arduino такие координаты:

```
1 Xa = 127 * 250 / 1366 = 23
2 Ya = 127 * 300 / 768 = 49
```

Координата X = 23 в шестнадцатеричном виде равна 0x17, а Y = 49 равна 0x31. Команда целиком будет выглядеть следующим образом:

```
1 0xDC 0x17 0x31 0x1
```

Листинг 5-6 демонстрирует управляющий скрипт для программы `mouse.ino`.

Листинг 5-6. Скрипт `ControlMouse.au3`

```
#include "CommInterface.au3"

func ShowError()
    MsgBox(16, "Error", "Error " & @error)
endfunc

func OpenPort()
    local const $iPort = 8
    local const $iBaud = 9600
    local const $iParity = 0
    local const $iByteSize = 8
    local const $iStopBits = 1

    $hPort = _CommAPI_OpenCOMPort($iPort, $iBaud, $iParity, $iByteSize, $iStopBits)
    if @error then
        ShowError()
        return NULL
    endif

    _CommAPI_ClearCommError($hPort)
    if @error then
        ShowError()
        return NULL
    endif

    _CommAPI_PurgeComm($hPort)
    if @error then
        ShowError()
        return NULL
    endif

    return $hPort
endfunc

func GetX($x)
    return (127 * $x / 1366)
endfunc

func GetY($y)
    return (127 * $y / 768)
endfunc

func SendArduino($hPort, $x, $y, $button)
    local $command[4] = [0xDC, GetX($x), GetY($y), $button]

    _CommAPI_TransmitString($hPort, _
        StringFromASCIIArray($command, 0, UBound($command), 1))

    if @error then ShowError()
endfunc

func ClosePort($hPort)
    _CommAPI_ClosePort($hPort)
    if @error then ShowError()
endfunc

$hWnd = WinGetHandle("[CLASS:MSPaintApp]")
WinActivate($hWnd)
Sleep(200)

$hPort = OpenPort()

SendArduino($hPort, 250, 300, 1)
```

```
64
65 ClosePort($hPort)
```

Этот скрипт очень похож на `ControlKeyboardCombo.au3` из листинга 5-4. Теперь в функцию `SendArduino` передаются четыре параметра: дескриптор порта, координаты курсора X и Y, код кнопки для нажатия. Кроме этого появились две новые функции: `GetX` и `GetY`. Они переводят соответствующие координаты из шкалы экрана в шкалу Arduino платы.

В функциях `GetX` и `GetY` используется текущее разрешение экрана. В нашем примере оно равно 1366×768. Не забудьте поменять его на актуальное значение для вашего монитора.

Для тестирования эмулятора мыши выполните следующие шаги:

1. Загрузите программу `mouse.ino` на Arduino плату.
2. Запустите приложение Paint. Переключитесь в нём на инструмент Brush (кисть).
3. Запустите скрипт `ControlMouse.au3`.

Скрипт симулирует щелчок левой кнопки мыши в точке с абсолютными координатами 250×300 в окне Paint. В ней должна появиться чёрная точка.

Эмуляция клавиатуры и мыши

Мы разработали программы для Arduino платы, чтобы эмулировать клавиатуру и мышь по отдельности. Такое решение хорошо работает, если для управления персонажем в игре требуется только одно из устройств ввода. Если же нужны оба, вам придётся купить две платы, запрограммировать их и сделать так, чтобы бот управлял обоими. Это неудобно. Намного лучше будет совместить функции эмуляции клавиатуры и мыши в одном устройстве. HID интерфейс это позволяет. Единственная сложность заключается в протоколе передачи данных по UART. Нам потребуется его расширить.

Прежде всего программа платы должна понять, какое именно действие требует выполнить управляющий AutoIt-скрипт. Назначим каждому из возможных действий код. Например, как предложено в таблице 5-4.

Таблица 5-4. Коды симулируемых действий

Код	Действие
0x1	Нажатие клавиши без модификатора.
0x2	Нажатие клавиши с модификатором.
0x3	Щелчок мыши.

В команде код действия должен идти сразу после байта преамбулы. Благодаря этому программа сможет правильно интерпретировать оставшиеся данные. Если код равен 0x1 или 0x2, применяется алгоритм симуляции нажатия клавиши из программы `keyboard-combo.ino` (листинг 5-3). В случае кода 0x3, отрабатывает алгоритм программы `mouse.ino` (листинг 5-5).

Листинг 5-7 демонстрирует программу для платы, которая поддерживает новый формат команд.

Листинг 5-7. Программа `keyboard-mouse.ino`

```
#include <Mouse.h>
#include <Keyboard.h>

void setup()
{
  Serial.begin(9600);
  Keyboard.begin();
  Mouse.begin();
}

void pressKey(char key)
{
  Keyboard.write(key);
}

void pressKey(char modifier, char key)
{
  Keyboard.press(modifier);
  Keyboard.write(key);
  Keyboard.release(modifier);
}

void click(signed char x, signed char y, char button)
{
  Mouse.move(x, y);
  Mouse.click(button);
}

void loop()
{
  static const char PREAMBLE = 0xDC;
  static const uint8_t BUFFER_SIZE = 5;
  enum
  {
    KEYBOARD_COMMAND = 0x1,
    KEYBOARD_MODIFIER_COMMAND = 0x2,
    MOUSE_COMMAND = 0x3
```

```
38   };
39
40   if (Serial.available() > 0)
41   {
42     char buffer[BUFFER_SIZE] = {0};
43     uint8_t readBytes = Serial.readBytes(buffer, BUFFER_SIZE);
44
45     if (readBytes != BUFFER_SIZE)
46       return;
47
48     if (buffer[0] != PREAMBLE)
49       return;
50
51     switch(buffer[1])
52     {
53       case KEYBOARD_COMMAND:
54         pressKey(buffer[3]);
55         break;
56
57       case KEYBOARD_MODIFIER_COMMAND:
58         pressKey(buffer[2], buffer[3]);
59         break;
60
61       case MOUSE_COMMAND:
62         click(buffer[2], buffer[3], buffer[4]);
63         break;
64     }
65   }
66 }
```

Для выбора симулируемого действия в зависимости от полученного кода, мы используем оператор `switch` в функции `loop`. Этот оператор проверяет значение второго байта команды. Он определяет, какая из функций будет вызвана для обработки оставшихся байт. Для удобства в операторе `switch` мы используем константы с кодами команд: `KEYBOARD_COMMAND` (0x1), `KEYBOARD_MODIFIER_COMMAND` (0x2) и `MOUSE_COMMAND` (0x3).

Возможно, вы заметили, что в случае команды на нажатие клавиши управляющий скрипт передаёт лишние данные. Метод `readBytes` объекта `Serial` всегда читает пять байтов (это константа `BUFFER_SIZE`) из входного буфера UART. Но используются из них только три в случае нажатия без модификатора или четыре – с модификатором. Можно ли оптимизировать эти накладные расходы и не передавать лишние данные? Предположим, что мы исправили управляющий скрипт. В результате этого длина команды зависит от кода действия, указанного во втором байте. Проблема в том, что мы должны передать в метод `readBytes` число байт для чтения из входного буфера UART. Но на момент его вызова, эта информация неизвестна. Поэтому нам придётся воспользоваться другим методом объекта `Serial`.

Метод `readBytesUntil` позволяет читать байты из входного буфера до тех пор, пока не встретится символ ограничитель или терминатор. Ограничитель – это предопределённое значение, которое сигнализирует об окончании передачи. Такой подход выглядит перспективным. Единственный вопрос, на который осталось ответить: какой ограничитель выбрать? Если вы задумаетесь над ним, то придёте к выводу, что однозначного ответа нет. Ограничитель, как и преамбула, – это один байт. Его значение не должно встречаться в данных команды. То есть отпадают все значения, которые могут принять координаты позиций курсора мыши (от 0x00 до 0x7F) и коды клавиш (от 0x00 до 0xFF). К сожалению, код клавиши может быть любым из диапазона значений, помещающихся в один байт. Поэтому нельзя гарантировать уникальность ограничителя. Мы могли бы увеличить его длину до двух байт. Это бы решило проблему, но тогда мы ничего не выиграем от команд переменной длины. Нам придётся передавать столько же байт, а иногда и больше, как и в случае с командами одинаковой длины.

Объект `Serial` предоставляет ещё один метод – `read`. Он читает все байты, находящиеся во входном буфере UART. С его помощью можно было бы решить нашу проблему, но только в том случае, если управляющий скрипт будет делать задержки между командами. Длительности каждой задержки должно быть достаточно, чтобы программа Arduino успела прочитать буфер. В противном случае в буфер будут попадать несколько команд за раз и различить их окажется проблематично. Этот подход ненадёжен, поскольку скрипт может генерировать запросы к плате очень часто.

В результате мы приходим к выводу, что накладные расходы, связанные с одинаковой длиной команд, приемлемы. Ими мы расплачиваемся за надёжную передачу данных.

Листинг 5-8 демонстрирует управляющий скрипт для программы `keyboard-mouse.ino`.

Листинг 5-8. Скрипт `ControlKeyboardMouse.au3`

```
#include "CommInterface.au3"

func ShowError()
    MsgBox(16, "Error", "Error " & @error)
endfunc

func OpenPort()
    local const $iPort = 10
    local const $iBaud = 9600
    local const $iParity = 0
```

```
    local const $iByteSize = 8
    local const $iStopBits = 1

    $hPort = _CommAPI_OpenCOMPort($iPort, $iBaud, $iParity, $iByteSize, $iStopBits)
    if @error then
        ShowError()
        return NULL
    endif

    _CommAPI_ClearCommError($hPort)
    if @error then
        ShowError()
        return NULL
    endif

    _CommAPI_PurgeComm($hPort)
    if @error then
        ShowError()
        return NULL
    endif

    return $hPort
endfunc

func SendArduinoKeyboard($hPort, $modifier, $key)
    if $modifier == NULL then
        local $command[5] = [0xDC, 0x1, 0xFF, $key, 0xFF]
    else
        local $command[5] = [0xDC, 0x2, $modifier, $key, 0xFF]
    endif

    _CommAPI_TransmitString($hPort, _
        StringFromASCIIArray($command, 0, UBound($command), 1))

    if @error then ShowError()
endfunc

func GetX($x)
    return (127 * $x / 1366)
endfunc

func GetY($y)
    return (127 * $y / 768)
endfunc

func SendArduinoMouse($hPort, $x, $y, $button)
    local $command[5] = [0xDC, 0x3, GetX($x), GetY($y), $button]

    _CommAPI_TransmitString($hPort, _
        StringFromASCIIArray($command, 0, UBound($command), 1))

    if @error then ShowError()
endfunc

func ClosePort($hPort)
    _CommAPI_ClosePort($hPort)
    if @error then ShowError()
endfunc

$hPort = OpenPort()

$hWnd = WinGetHandle("[CLASS:MSPaintApp]")
WinActivate($hWnd)
Sleep(200)
```

```
75
76 SendArduinoMouse($hPort, 250, 300, 1)
77
78 Sleep(1000)
79
80 $hWnd = WinGetHandle("[CLASS:Notepad]")
81 WinActivate($hWnd)
82 Sleep(200)
83
84 SendArduinoKeyboard($hPort, Null, 0x54) ; T
85 SendArduinoKeyboard($hPort, Null, 0x65) ; e
86 SendArduinoKeyboard($hPort, Null, 0x73) ; s
87 SendArduinoKeyboard($hPort, Null, 0x74) ; t
88
89 Sleep(1000)
90
91 SendArduinoKeyboard($hPort, 0x82, 0xB3) ; Alt+Tab
92
93 ClosePort($hPort)
```

В этом скрипте мы реализовали две отдельные функции для симуляции действий клавиатуры и мыши. `SendArduinoKeyboard` отправляет на плату команду для нажатия клавиши. Её алгоритм почти такой же, как у функции `SendArduino` из скрипта `ControlKeyboardCombo.au3` (листинг 5-4). Отличие в формате команды: появился второй байт с кодом действия. Также мы дополняем байтовый массив на выдачу до необходимой длины в пять байтов с помощью константного значения 0xFF. Если нажатие симулируется без модификатора, то третий байт сообщения также заменяется на 0xFF.

Функция `SendArduinoMouse` отправляет команду для симуляции щелчка мыши. Единственное её отличие от аналога из скрипта `ControlMouse.au3` (листинг 5-6) – добавлен код действия во втором байте.

Чтобы протестировать скрипт `ControlKeyboardMouse.au3`, выполните следующие действия:

1. Загрузите программу `keyboard-mouse.ino` на Arduino-плату.
2. Запустите приложение Paint.
3. Запустите приложение Notepad.
4. Запустите скрипт.

Скрипт последовательно выполнит три действия:

1. Щелчок левой кнопкой мыши в окне Paint.
2. Набор строки “Test” в окне Notepad.

3. Переключение между открытыми окнами с помощью комбинации клавиш Alt+Tab.

Может возникнуть вопрос: почему мы использовали константное значение 0xFF для дополнения команд до нужной длины? Разумнее было бы подставлять 0x00. Это решение продиктовано особенностью AutoIt-функции `StringFromASCIIArray`, с помощью которой мы конвертируем массив в строку. Она обрабатывает значение 0x00 как ограничитель строки. Другими словами, результирующая строка будет обрезана до этого символа. Эта особенность означает, что все наши команды не должны содержать нулевых байтов. Следовательно, мы не сможем симулировать нажатие клавиши с кодом 0x00.

Выводы

Мы рассмотрели технику эмуляции клавиатуры и мыши с помощью платы Arduino. AutoIt скрипт, в котором реализована вся логика бота, может управлять ею через UART интерфейс. Таким образом совмещаются возможности анализа изображения на экране и симуляции действий устройств ввода. Благодаря этому вашего кликера будет невозможно обнаружить с помощью защит, основанных на проверке состояния клавиатуры и мыши.

Перехват данных на уровне ОС

В третьей главе мы рассмотрели методы чтения состояний объектов из памяти процесса игрового приложения. Хорошо продуманная защита может значительно усложнить их применение. В этом случае имеет смысл попробовать альтернативный подход, который заключается в подмене или модификации системных библиотек. Это позволит вам изменить точку перехвата данных. Теперь состояния объектов будут читаться не из памяти процесса, а из используемых им DLL-библиотек. Проконтролировать их намного труднее. Высока вероятность, что система защиты с этим не справится.

Инструменты для разработки

Нам предстоит активная работа с WinAPI-функциями и системными библиотеками. Для этой задачи лучше всего подойдёт язык C++. Для

компиляции примеров воспользуемся Visual Studio IDE. Инструкцию по её установке вы найдёте в третьей главе.

Есть несколько решений с открытым исходным кодом для перехвата вызовов WinAPI. Первое из них называется **DLL Wrapper Generator** (генератор обёрток DLL). Мы будем использовать его, чтобы создавать обёртки для системных библиотек.

Для установки генератора выполните следующие шаги:

1. Скачайте архив со скриптами со страницы проекта на Github.
2. Скачайте и установите Python версии 2.7.

Второе решение, которым мы воспользуемся, называется **Deviare**. Это **фреймворк** для перехвата вызовов DLL-библиотек.

Чтобы установить Deviare, сделайте следующее:

1. Скачайте архив с уже собранными исполняемыми файлами и библиотеками фреймворка.
2. Скачайте архив с исходным кодом той же версии.
3. Распакуйте оба архива в разные каталоги.

Список сборок Deviare доступен на Github странице проекта. Ещё раз проверьте, что версии скачанной сборки и исходного кода совпадают.

Тестовое приложение

Чтобы продемонстрировать методы перехвата WinAPI-вызовов, понадобится какое-то целевое приложение. Предлагаю воспользоваться программой, разработанной нами в разделе “Методы защиты от внутриигровых ботов” третьей главы. Немного изменённая версия её исходного кода приведена в листинге 5-9.

Листинг 5-9. Исходный код тестового приложения

```
#include <stdio.h>
#include <stdint.h>
#include <windows.h>
#include <string>

static const uint16_t MAX_LIFE = 20;
volatile uint16_t gLife = MAX_LIFE;

int main()
{
```

```
11      SHORT result = 0;
12
13      while (gLife > 0)
14      {
15          result = GetAsyncKeyState(0x31);
16          if (result != 0xFFFF8001)
17              --gLife;
18          else
19              ++gLife;
20
21          std::string str(gLife, '#');
22          TextOutA(GetDC(NULL), 0, 0, str.c_str(), str.size());
23
24          printf("life = %u\n", gLife);
25          Sleep(1000);
26      }
27      printf("stop\n");
28      return 0;
29  }
```

Алгоритм работы приложения не изменился. Каждую секунду значение глобальной переменной `gLife` уменьшается на единицу, если клавиша “1” не была нажата. В противном случае `gLife` увеличивается на один. Теперь вместо вывода на консоль с помощью функции `printf`, мы делаем WinAPI-вызов `TextOutA`. Он печатает строку, переданную в качестве входного параметра, в левом верхнем углу экрана. В нашем случае строка состоит из символов решётки, число которых соответствует значению переменной `gLife`.

Зачем мы изменили функцию вывода информации? Наша цель заключается в перехвате WinAPI-вызовов. Функция `printf` предоставляется не WinAPI, а **библиотекой времени выполнения** языка C. В этой библиотеке реализованы низкоуровневые функции, описанные в стандарте языка. Доступ к ним возможен как из приложений, написанных на C, так и C++. Конечно, техника перехвата вызовов подойдёт и для случая с `printf`. Но для примера будет интереснее разобрать вариант именно с WinAPI-функцией. Поэтому мы используем `TextOutA`.

Согласно документации WinAPI, функция `TextOutA` реализована в системной библиотеке `gdi32.dll`. Эта информация пригодится нам в дальнейшем.

Скомпилируйте приложение на Visual Studio под 32-разрядную платформу и запустите, чтобы проверить его работу.

Загрузка DLL-библиотек

Перед тем как разбираться с техниками перехвата WinAPI-вызовов, рассмотрим взаимодействие приложения и используемой им DLL-библиотеки.

Когда мы запускаем какое-то приложение, **загрузчик программ Windows** (PE-загрузчик) читает содержимое исполняемого файла в оперативную память. Точнее в область памяти нового процесса. Загруженный код называется **EXE-модулем**. Стандартным форматом исполняемых файлов в Windows является **PE**. Он определяет структуру данных (известную как **PE-заголовок**), которая хранится в начале файла. Она содержит всю необходимую информацию для запуска приложения. Список используемых DLL библиотек является её частью.

На следующем шаге PE-загрузчик ищет файлы необходимых DLL-библиотек на жёстком диске. Их содержимое читается с диска и записывается в память процесса запускаемого приложения. Загруженный код одной библиотеки называется **DLL модулем**. Было бы логично размещать DLL-модули по одним и тем же адресам при каждом запуске приложения. К сожалению, всё не так просто. Эти адреса выбираются случайно механизмом Windows под названием **Address Space Load Randomization** (ASLR). Он защищает ОС от некоторых видов вредоносного ПО. Минус такого подхода в том, что компилятор не может использовать статические адреса для вызова функций библиотек из EXE модуля.

Проблема решается с помощью **Import Table** (таблица импорта). Кроме неё есть так называемая **Thunk Table** (таблица переходов). Эти таблицы часто путают. Рассмотрим подробнее их внутреннее устройство.

Import Table представляет собой массив структур типа `IMAGE_IMPORT_DESCRIPTOR`:

```
1 typedef struct _IMAGE_IMPORT_DESCRIPTOR {
2     DWORD   OriginalFirstThunk;
3     DWORD   TimeDateStamp;
4     DWORD   ForwarderChain;
5     DWORD   Name;
6     DWORD   FirstThunk;
7 } IMAGE_IMPORT_DESCRIPTOR, *PIMAGE_IMPORT_DESCRIPTOR;
```

Каждая такая структура соответствует одной DLL-библиотеке. В поле `Name` хранится имя её файла. Число `OriginalFirstThunk` на самом деле является

указателем на первый элемент массива структур типа `IMAGE_THUNK_DATA`:

```
typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD    Hint;
    BYTE    Name[1];
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;

typedef struct _IMAGE_THUNK_DATA {
    union {
        PDWORD                  Function;
        PIMAGE_IMPORT_BY_NAME  AddressOfData;
    } u1;
} IMAGE_THUNK_DATA, *PIMAGE_THUNK_DATA;
```

Ключевое слово `union` в определении `IMAGE_THUNK_DATA` говорит о том, что данные могут интерпретироваться двумя способами:

1. Как указатель типа `PDWORD` на функцию в памяти запущенного процесса.
2. Как указатель на структуру типа `IMAGE_IMPORT_BY_NAME`, которая содержит порядковый номер функции в библиотеке и её символьное имя.

Поле `FirstThunk` структуры `IMAGE_IMPORT_DESCRIPTOR` указывает на первый элемент массива, известного как **Import Address Table** (таблица импорта адресов) или IAT. PE-загрузчик перезаписывает её адресами функций из соответствующей загруженной DLL-библиотеки. Более подробно структура Import Table описана в русской и английской статьях.

Import Table является частью PE-заголовка. В ней хранится общая информация о требуемых DLL библиотеках. Всё содержимое PE-заголовка загружается в сегмент памяти процесса с правами только на чтение. Thunk Table является частью исполняемого кода. Она содержит переходы (**thunk**) на импортируемые функции. Эти переходы представляют собой ассемблерные инструкции `JMP`. Thunk Table загружается в `.text` сегмент с правами на чтение и исполнение. В этом же сегменте хранится исполняемый код приложения. Import Address Table, на которую указывает `FirstThunk` элементов Import Table, помещается в сегмент `.idata` с правами на чтение и запись.

Некоторые компиляторы генерируют код, не использующий Thunk Table. Благодаря этому удаётся избежать накладных расходов, связанных с дополнительным `JMP` переходом. Код, сгенерированный компилятором MinGW, использует Thunk Table. В этом случае схема вызова

импортируемой функции `TextOutA` будет соответствовать иллюстрации 5-1.

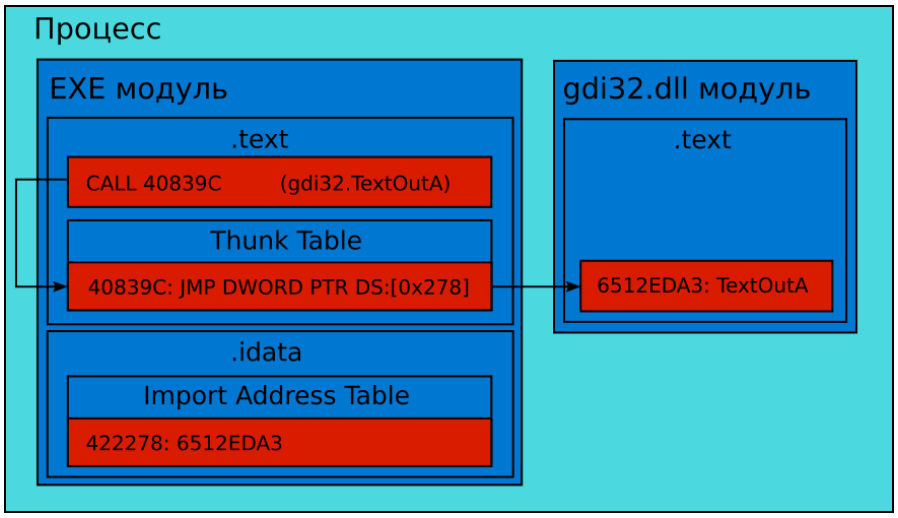


Иллюстрация 5-1. Вызов функции `TextOutA` из приложения, скомпилированного MinGW

Алгоритм вызова выглядит следующим образом:

1. Процессор переходит к инструкции ассемблера `CALL 40839C`. Она выполняет вызов функции. При этом адрес возврата из неё помещается в стек, а управление передаётся элементу Thunk Table по адресу 40839C.
2. Элемент Thunk Table содержит единственную инструкцию `JMP`. Она выполняет безусловный переход на функцию `TextOutA` модуля `gdi32.dll`, загруженном в память исполняемого процесса. Линейный адрес функции извлекается из Import Address Table. Для доступа к ней используется регистр DS, указывающий на сегмент `.idata`. Для расчёта адреса элемента Import Address Table используется сдвиг (в нашем случае равный 0x278):

```
1 DS + 0x278 = 0x422000 + 0x278 = 0x422278
```

3. Процессор выполняет код `TextOutA`. Последняя инструкция функции — это `RETN`. Она извлекает адрес возврата из стека и осуществляет переход на инструкцию, следующую сразу за `CALL` в EXE модуле, откуда начинался вызов.

Компилятор Visual C++ генерирует код, который не использует Thunk Table. Схема вызова функции `TextOutA` в этом случае выглядит как на иллюстрации 5-2. Алгоритм этого вызова следующий:

1. Процессор выполняет инструкцию ассемблера `CALL DWORD PTR DS:[0x10C]`. В ней происходит чтение линейного адреса функции из Import Address Table. Затем на стек помещается адрес возврата. После этого управление передаётся в функцию `TextOutA` модуля `gdi32.dll`.
2. Процессор выполняет код `TextOutA`. Возврат из неё в EXE-модуль происходит по инструкции `RETN`.

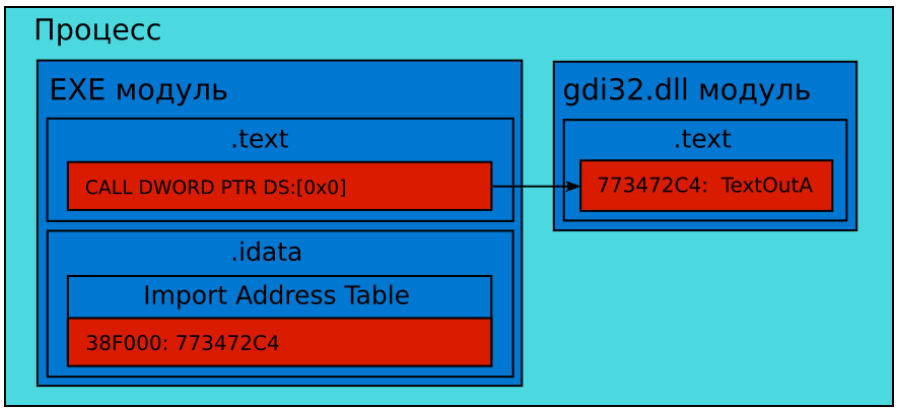


Иллюстрация 5-2. Вызов функции `TextOutA` из приложения, скомпилированного Visual C++

Техники перехвата вызовов WinAPI

Игровые приложения взаимодействуют с ОС и её ресурсами через системные DLL-библиотеки. Например, чтобы вывести на экран текст, вызывается функция `TextOutA` или аналогичная. Перехватив этот вызов, мы узнаем текст, который приложение пытается вывести. Такой подход чем-то напоминает перехват данных устройства вывода. Только теперь мы получаем эти данные до того, как они будут отображены на экране.

Инструмент API Monitor, применявшийся во второй главе, хорошо демонстрирует принцип перехвата вызовов WinAPI. Все функции WinAPI, которые вызывал анализируемый процесс, выводятся в окне приложения “Summary”. Мы можем реализовать бота, который будет вести себя как API Monitor. Но вместо вывода перехваченных вызовов, он должен симулировать действия игрока.

Рассмотрим на примерах две наиболее известные и используемые техники перехвата вызовов.

Proxy DLL

Идея первой техники заключается в подмене Windows-библиотеки. Мы могли бы подготовить DLL-библиотеку, которая выглядит так же как системная с точки зрения PE-загрузчика. В этом случае она будет загружена в память процесса приложения. Игра будет взаимодействовать с подложной библиотекой точно так же, как если бы это была системная. Благодаря этому наш код будет получать управление при каждом вызове функции из неё. Подложная библиотека называется **proxy DLL**.

В большинстве случаев надо перехватывать несколько определённых вызовов WinAPI. Все остальные функции замещаемой системной библиотеки нам не интересны и должны работать как обычно. Кроме того при подмене DLL-библиотек помните важное правило: процесс должен вести себя с proxy DLL точно так же, как и с оригинальной библиотекой. В противном случае нельзя гарантировать его корректную работу. Эти два обстоятельства наводят на мысль, что proxy DLL должна уметь перенаправлять в оригинальную библиотеку все вызовы приложения.

Когда процесс игрового приложения вызывает функцию из proxy DLL, наш код получает управление. Он может симулировать действия пользователя или просто читать для бота состояния игровых объектов. После этого обязательно надо передать управление WinAPI-функции, выполнение которой ожидает приложение. В противном случае оно просто завершится с ошибкой или продолжит работу в не консистентном состоянии, то есть его данные окажутся не согласованы.

Итак, если мы не собираемся перехватывать какую-то функцию WinAPI, мы просто перенаправляем её вызов в системную библиотеку. В противном случае сначала отрабатывает наш код, и только потом управление передаётся в системную библиотеку. Это означает, что она должна быть загружена в адресное пространство процесса. Иначе код оригинальных WinAPI-функций будет недоступен. Очевидно, что PE-загрузчик ничего не знает про замещённую библиотеку. Он загрузил proxy DLL, и на этом его работа выполнена. Оригинальную библиотеку должна загружать proxy DLL с помощью WinAPI-функции `LoadLibrary`.

Иллюстрация 5-3 демонстрирует схему вызова WinAPI-функции `TextOutA` через proxy DLL в случае компиляции приложения на Visual C++.

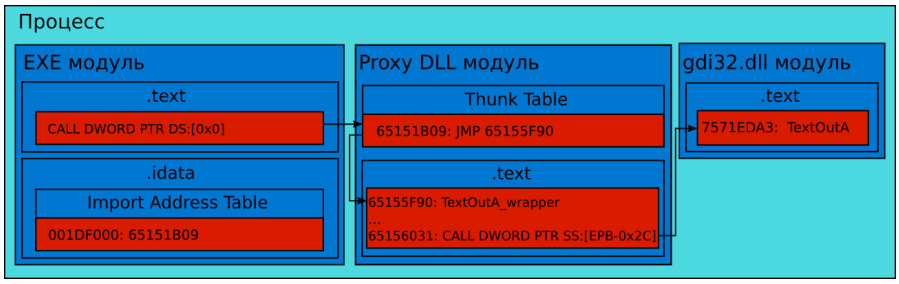


Иллюстрация 5-3. Вызов функции `TextOutA` через proxy DLL

Алгоритм вызова функции следующий:

1. PE-загрузчик загружает proxy DLL вместо системной библиотеки `gdi32.dll`. При этом он записывает линейные адреса всех функций, экспортируемых proxy DLL, в Import Address Table модуля EXE.
2. Исполнение кода модуля EXE достигает точки вызова функции `TextOutA`. Дальше отрабатывает стандартный алгоритм вызова функции из импортируемой DLL. Инструкция `CALL` сохраняет адрес возврата на стеке и передаёт управление по адресу из Import Address Table. Единственное отличие в том, что управление получает не системная библиотека, а подменяющая её proxy DLL.
3. В proxy DLL есть Thunk Table, элемент которой получает управление от `CALL` инструкции EXE-модуля. Именно линейные адреса элементов Thunk Table записываются PE-загрузчиком в Import Address Table модуля EXE.
4. Инструкция `JMP` элемента Thunk Table выполняет безусловный переход в обёртку для функции `TextOutA` под названием `TextOutA_wrapper`, которая реализована в proxy DLL. В ней отрабатывает код бота.
5. В конце кода обёртки находится инструкция `CALL`, которая сохраняет на стеке адрес возврата и передаёт управление оригинальной функции `TextOutA` из модуля `gdi32.dll`.
6. После отработки оригинальной функции `TextOutA`, она возвращает управление в обёртку `TextOutA_wrapper` через инструкцию `RETN`.
7. Инструкция `RETN` в обёртке возвращает управление обратно в EXE-модуль.

Может возникнуть вопрос: как proxy DLL узнаёт линейные адреса функций, которые экспортируются системной библиотекой `gdi32.dll`? В обычной ситуации эти адреса читаются PE-загрузчиком. Но сейчас мы не можем его задействовать, ведь он загружает только proxy DLL. Опять же эта задача должна выполняться proxy DLL самостоятельно. WinAPI-вызов `GetProcAddress` возвращает линейный адрес функции из указанного модуля по её имени или порядковому номеру.

Ещё один момент остаётся неясным. Что нужно сделать, чтобы PE-загрузчик выбрал proxy DLL вместо системной библиотеки? У Windows есть механизм поиска динамических библиотек. Пути всех системных DLL хранятся в **реестре** и только по ним происходит поиск. Мы не можем просто подменить системную библиотеку в каталоге `Windows`. Скорее всего, её используют многие сервисы и службы ОС. Велика вероятность, что система окажется неработоспособной после такой подмены. Кроме того, оригинальная библиотека должна храниться в месте, известном всем её клиентам, поскольку все вызовы proxy DLL должны перенаправляться в неё. Правильное решение заключается в том, чтобы поместить proxy DLL в каталог приложения, вызовы которого мы собираемся перехватывать. Чтобы механизм защиты Windows не мешал загрузке библиотеки, его надо отключить. Для этого достаточно отредактировать реестр.

Преимущества использования proxy DLL перед другими техниками перехвата WinAPI-вызовов следующие:

1. Очень просто сгенерировать proxy DLL с помощью существующих бесплатных утилит.
2. Подмена библиотеки происходит только для одного конкретного приложения. Все остальные системные сервисы и службы используют оригинальную DLL.
3. Защитить приложение от такого перехвата вызовов может быть сложно.

Недостатки подхода proxy DLL:

1. Некоторые системные библиотеки невозможно подменить (например `kernel32.dll`). Причина этого ограничения в том, что обе WinAPI-функции `LoadLibrary` и `GetProcAddress` предоставляются `kernel32.dll`. Это значит, что они должны быть доступны в момент, когда proxy DLL загружает системную библиотеку.

Пример использования proxy DLL

Применим технику перехвата WinAPI-вызовов с помощью proxy DLL на практике. Напишем бота для нашего тестового приложения, который будет поддерживать значение `gLife` больше десяти. Для простоты встроим алгоритм бота в код proxy DLL.

Первая задача заключается в том, чтобы сгенерировать proxy DLL с заглушками для каждой WinAPI-функции из замещаемой системной библиотеки `gdi32.dll`. В этом нам поможет скрипт DLL Wrapper Generator. Для его запуска выполните следующие шаги:

1. Скопируйте 32-разрядную версию системной библиотеки `gdi32.dll` в каталог скрипта-генератора. Она находится в каталоге `C:\Windows\system32` на 32-разрядной Windows или в `C:\Windows\SysWOW64` для 64-разрядной.
2. Запустите скрипт-генератор из командной строки CMD:

```
1 python Generate_Wrapper.py gdi32.dll
```

Будет создан Visual Studio проект с исходным кодом proxy DLL в подкаталоге `gdi32`.

Во всех примерах мы будем использовать 32-разрядную версию тестового приложения.

Теперь реализуем алгоритм бота в сгенерированной proxy DLL. Для этого выполните следующее:

1. В Visual Studio откройте файл проекта gdi32. Его формат устарел, поэтому Visual Studio предложит обновление до актуальной версии. Для этого нажмите кнопку “OK” в диалоге “Upgrade VC++ Compiler and Libraries” (обносить компилятор VC++ и библиотеки).
2. В файле проекта `gdi32.cpp` измените путь до системной библиотеки `gdi32.dll` в вызове `LoadLibrary`. Вам нужна строчка номер 10, которая выглядит следующим образом:

```
1 mHinstDLL = LoadLibrary( "ori_gdi32.dll" );
```

Замените строку “ori_gdi32.dll” на корректный путь библиотеки в вашей системе. В случае 64-разрядной Windows должно получиться следующее:

```
mHinstDLL = LoadLibrary( "C:\\Windows\\SysWOW64\\gdi32.dll" );
```

3. В том же файле `gdi32.cpp` замените обёртку функции `TextOutA` с именем `TextOutA_wrapper` на код из листинга 5-10.

Листинг 5-10. Алгоритм бота, реализованный в обёртке функции `TextOutA`

```
extern "C" BOOL __stdcall TextOutA_wrapper(
    _In_ HDC     hdc,
    _In_ int     nXStart,
    _In_ int     nYStart,
    _In_ LPCSTR lpString,
    _In_ int     cchString
    )
{
    if (cchString < 10)
    {
        INPUT Input = { 0 };
        Input.type = INPUT_KEYBOARD;
        Input.ki.wVk = '1';
        SendInput(1, &Input, sizeof(INPUT));
    }

    typedef BOOL(__stdcall *pS)(HDC, int, int, LPCTSTR, int);
    pS pps = (pS)mProcs[696];
    return pps(hdc, nXStart, nYStart, lpString, cchString);
}
```

Полная версия файла `gdi32.cpp` доступна в архиве с примерами к этой книге.

Вспомним код вызова функции `TextOutA` из нашего тестового приложения:

```
std::string str(gLife, '#');
TextOutA(GetDC(NULL), 0, 0, str.c_str(), str.size());
```

Здесь мы используем объект `string` **библиотеки STL**. Его **конструктор** принимает два входных параметра: длину строки и символ, которым её надо заполнить. В качестве длины мы передаём переменную `gLife`. Дальше объект `string` используется в вызове `TextOutA`. Параметры этой WinAPI-функции приведены в таблице 5-5.

Таблица 5-5. Параметры WinAPI-функции `TextOutA`

Номер параметра	Переданное значение	Описание
1	GetDC(NULL)	Контекст устройства в котором будет напечатана строка.
2	0	Координата X начала строки.
3	0	Координата Y начала строки.
4	str.c_str()	Указатель на строку с нуль-символом на конце.
5	str.size()	Длина строки в байтах.

Алгоритм бота выглядит следующим образом:

1. Последний параметр с именем `cchString` обёртки `TextOutA_wrapper` хранит длину выводимой строки. Эта длина равна переменной `gLife` тестового приложения. Сравниваем её со значением 10.
2. Если длина строки меньше десяти, симулируем нажатие клавиши “1” с помощью WinAPI-функции `SendInput`. В противном случае ничего не делаем.
3. Вызываем функцию `TextOutA` из системной библиотеки `gdi32`. Для этого используем указатель на неё, хранящийся в глобальном массиве `mProcs`. Он содержит указатели на все функции, экспортируемые библиотекой `gdi32.dll`. Его инициализация происходит в функции `DllMain` в момент загрузки proxy DLL в память процесса (см листинг 5-11).

Листинг 5-11. Инициализация массива `mProcs` с указателями на функции `gdi32.dll`

```
HINSTANCE mHinst = 0, mHinstDLL = 0;
UINT_PTR mProcs[727] = {0};
LPCSTR mImportNames[] = {...}

BOOL WINAPI DllMain( HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved ) {
    mHinst = hinstDLL;
    if ( fdwReason == DLL_PROCESS_ATTACH ) {
        mHinstDLL = LoadLibrary( "C:\\Windows\\SysWOW64\\gdi32.dll" );
        if ( !mHinstDLL )
            return ( FALSE );
        for (int i = 0; i < 727; i++)
        {
            mProcs[i] = (UINT_PTR)GetProcAddress(mHinstDLL, mImportNames[i]);
        }
    } else if ( fdwReason == DLL_PROCESS_DETACH ) {
        FreeLibrary( mHinstDLL );
    }
    return ( TRUE );
}
```

Алгоритм инициализации массива `mProcs` крайне прост. Скрипт-генератор составил список имён экспортируемых библиотекой функций и поместил его в массив `mImportNames`. В функции `DllMain` мы загружаем `gdi32.dll` библиотеку с помощью WinAPI-вызова `LoadLibrary`. Затем циклом `for` проходим по массиву `mImportNames` и для каждого имени функции читаем её адрес с помощью `GetProcAddress`. Результат сохраняем массив `mProcs`.

Как в листинге 5-10 мы узнали, что порядковый номер `TextOutA` в массиве `mProcs` равен 696? Этот номер указан в обёртке, которую сгенерировал скрипт DLL Wrapper Generator:

```
1 extern "C" __declspec(naked) void TextOutA_wrapper(){__asm{jmp mProcs[696*4]}}
```

Единственный неясный момент: почему в сгенерированной обёртке индекс 696 умножается на 4? Дело в том, что в языке ассемблера любой массив представляется как байтовый. Однако, каждый элемент массива `mProcs` имеет тип `UINT_PTR`. Это указатель на беззнаковое целое. Размер всех указателей на 32-разрядной платформе равен четырём байтам (или 32 битам). Таким образом, если мы хотим из ассемблера получить доступ к элементу массива `mProcs` с индексом 696, мы должны умножить это число на размер элемента (т.е. на четыре). Язык C++ учитывает размер типа `UINT_PTR` и смещается на нужный элемент без дополнительного умножения.

Наша библиотека proxy DLL почти готова. Последние несколько шагов нужны, чтобы подготовить окружение для её использования:

1. Скомпилируйте проект gdi32 в Visual Studio под 32-разрядную архитектуру.
2. Скопируйте собранную proxy DLL с именем `gdi32.dll` в каталог с тестовым приложением `TestApplication.exe`.
3. Добавьте библиотеку `gdi32.dll` в ключ системного реестра `ExcludeFromKnownDLL`. Для этого через меню Start (Пуск) запустите стандартное Windows-приложение `regedit`. Путь до нужного ключа следующий:

```
1 HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\
2 Session Manager\ExcludeFromKnownDlls
```

4. Перезагрузите компьютер, чтобы изменения реестра вступили в силу.

Чего мы добились этой правкой реестра? В Windows есть механизм, защищающий системные библиотеки от подмены вредоносным ПО. Наиболее важные из них указываются в реестре. Таким образом PE-загрузчик загружает эти библиотеки только из предопределённых путей. Однако, есть специальный **ключ реестра** `ExcludeFromKnownDLL`, который отменяет эту защиту для указанных в нём библиотек. В него надо добавить `gdi32.dll`. После этого PE-загрузчик станет придерживаться стандартной последовательности поиска DLL библиотеки, начиная с текущего каталога запускаемого приложения. Таким образом, будет загружена proxy DLL.

Теперь вы можете запустить наше тестовое приложение. В окне консоли вы увидите, что параметр `gLife` не опускается ниже 10, благодаря действиям бота.

Модификация WinAPI

Вторая техника перехвата вызовов приложения, которую мы рассмотрим, заключается в модификации системных функций. Предположим, что PE-загрузчик прочитал `gdi32.dll` библиотеку в память процесса. Теперь, получив доступ к этой памяти, мы можем модифицировать функции `gdi32.dll` модуля, которые следует перехватить. Достаточно изменить только первую ассемблерную инструкцию, заменив её на переход в наш код.

Есть несколько способов передачи управления из WinAPI-функции. Самое распространённое решение заключается в использовании ассемблерных инструкций `JMP` или `CALL`. Таким образом код бота получит управление. После выполнения его алгоритма, мы должны вернуться в оригинальную WinAPI-функцию. Но после модификации, этого сделать нельзя. Мы получим **рекурсию**, поскольку бот будет циклично вызывать WinAPI-функцию, а она — его. Это приведёт к переполнению стека, поскольку в нём сохраняется адрес возврата. В результате приложение завершит свою работу с ошибкой. Чтобы предотвратить этот сценарий, нам следует восстановить первую инструкцию WinAPI-функции и только потом её вызывать. Когда она закончит свою работу, надо снова установить переход (`JMP` или `CALL`) на код бота. Так мы будем готовы к следующему вызову.

Каким образом можно модифицировать WinAPI-функции в памяти процесса? В третьей главе при разработке бота для Diablo 2 мы

рассмотрели способы записи в память. Но тогда речь шла о сегменте данных, который доступен для чтения и записи. Теперь же нам надо модифицировать сегмент кода с доступом на чтение и исполнение. Нам на помощь опять приходит WinAPI, который предоставляет функции `VirtualQuery` и `VirtualProtect`. С их помощью можно поменять флаги доступа к сегменту. Примеры использования этих функций приведены на форуме.

Мы разобрались, как модифицировать WinAPI-функции и передавать управление в код бота. Но остаётся ещё один вопрос. Чтобы код бота получил управление, он должен находиться в памяти процесса. Кто будет его загружать в нашем случае? PE-загрузчик и игровое приложение исключаются. Значит, это должен сделать сам бот с помощью WinAPI-функции `RemoteLoadLibrary`. Подробнее её использование описано в статье.

Иллюстрация 5-3 демонстрирует порядок вызова функции `TextOutA` после модификации WinAPI. В рассмотренном случае алгоритмы бота реализованы в библиотеке `handler.dll`.

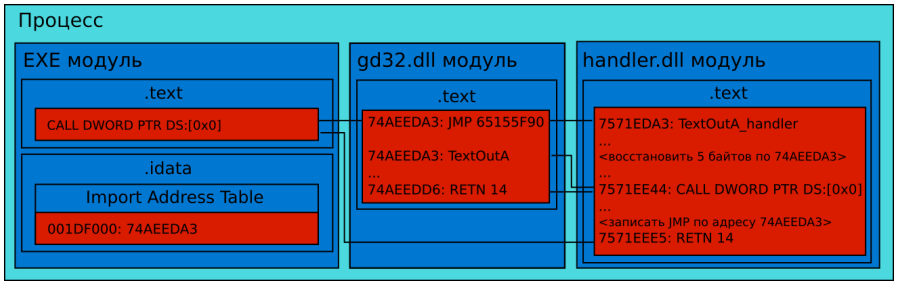


Иллюстрация 5-4. Вызов функции `TextOutA` после модификации WinAPI

Алгоритм вызова выглядит следующим образом:

1. С помощью WinAPI-функции `RemoteLoadLibrary` библиотека `handler.dll` загружается в память целевого процесса. Сразу после этого её функция `DllMain` получает управление и модифицирует функцию `TextOutA` в загруженном ранее модуле `gdi32.dll`.
2. Исполнение кода EXE-модуля достигает инструкции `CALL DWORD PTR DS:[0x0]`. В ней читается линейный адрес функции из Import Address

Table. Затем управление передаётся в функцию `TextOutA` модуля `gdi32.dll`.
3. Первая инструкция функции `TextOutA` заменена на `JMP` инструкцию. Она выполняет безусловный переход в обработчик `TextOutA_handler` модуля `handler.dll`.
4. Код бота отрабатывает в обработчике `TextOutA_handler`.
5. Обработчик `TextOutA_handler` восстанавливает в исходное значение первую инструкцию функции `TextOutA` модуля `gdi32.dll`. Затем она вызывается с помощью инструкции `CALL`.
6. После выполнения функция `TextOutA` возвращает управление обратно в обработчик `TextOutA_handler` с помощью инструкции `RETN`.
7. Первая инструкция `TextOutA` снова замещается на `JMP`, которая передаёт управление в модуль `handler.dll`.
8. Обработчик `TextOutA_handler` возвращает управление в EXE-модуль с помощью `RETN` на инструкцию следующую за вызовом `TextOutA`.

Техника модификации WinAPI имеет следующие достоинства:

1. Она позволяет перехватывать вызовы функций любой системной библиотеки (в том числе `kernel32.dll`).
2. Существует несколько фреймворков для модификации WinAPI. Они предоставляют в готовом виде большую часть кода для внедрения модуля DLL с обработчиком и модификации первой инструкции перехватываемой функции.

К недостаткам техники можно отнести:

1. Она не позволяет перехватывать вызовы функций, размер кода которых меньше пяти байтов. Это ограничение продиктовано размером инструкции `JMP`. Если функция короче этой инструкции, то её модификация может привести к завершению работы процесса с ошибкой.
2. Достаточно сложно реализовать эту технику вручную без использования фреймворков.
3. Техника работает ненадёжно с многопоточными приложениями. Причина заключается в том, что вызовы модифицированной WinAPI-функции никак не синхронизированы. Если она вызывается из первого потока (при этом первая инструкция функции

восстанавливается), то её вызовы из других потоков не будут перехвачены.

Пример модификации WinAPI

Разработаем бота, который использует технику модификации WinAPI. Он будет работать по хорошо знакомому нам алгоритму: симулировать нажатие кнопки “1”, если значение `gLife` опустится ниже 10. Для разработки мы воспользуемся фреймворком Deviare.

Сначала познакомимся с фреймворком и его основными возможностями. В архиве с ним распространяется несколько демонстрационных примеров. Один из них под названием **CTest** перехватывает WinAPI-вызовы и записывает информацию о них в текстовый файл. В этом примере реализованы основные шаги техники перехвата: загрузка DLL библиотеки с обработчиками вызовов в память целевого процесса и алгоритм модификации WinAPI-функций.

Попробуем перехватить вызовы нашего тестового приложение с помощью примера CTest. Для этого выполните следующие действия:

1. Скачайте архив с уже собранными исполняемыми файлами и библиотеками фреймворка. Распакуйте его в каталог с именем `deviare-bin`.
2. Скопируйте исполняемый файл тестового приложения `TestApplication.exe` в каталог `deviare-bin`.
3. Откройте для редактирования конфигурационный файл `ctest.hooks.xml` из каталога `deviare-bin`. В нём указаны WinAPI-вызовы, которые будут перехвачены. Добавьте в этот список функцию `TextOutA`:

```
1 <hook name="TextOutA">gdi32.dll!TextOutA</hook>
```

4. В командной строке запустите пример `CTest` со следующими параметрами:

```
1 CTest.exe exec TestApplication.exe -log=out.txt
```

Рассмотрим параметры командной строки примера `CTest.exe`. Первый из них `exec TestApplication.exe` указывает целевое приложение, которое следует запустить. После запуска в память процесса TestApplication будет

загружена DLL библиотека с обработчиками вызовов. Второй параметр `-log=out.txt` указывает текстовый файл для вывода информации о перехваченных вызовах.

После запуска откроются два окна: CTest и TestApplication. Когда значение переменной `gLife` достигнет нуля в окне TestApplication, остановите выполнение приложения CTest нажатием Ctrl+C в его окне.

Откройте лог-файл `out.txt` и найдите в нём следующие строчки:

```
CNktDvEngine::CreateHook (gdi32.dll!TextOutA) => 00000000
...
21442072: Hook state change [2500]: gdi32.dll!TextOutA -> Activating
...
21442306: LoadLibrary [2500]: C:\Windows\System32\gdi32.dll / Mod=00000003
...
21442852: Hook state change [2500]: gdi32.dll!TextOutA -> Active
```

Они означают, что CTest успешно модифицировал WinAPI-функцию `TextOutA` модуля `gdi32.dll` в памяти тестового приложения. Прокрутите лог-файл дальше. Вы найдёте информацию о каждом перехваченном вызове `TextOutA` в следующем виде:

```
21442852: Hook called [2500/2816 - 1]: gdi32.dll!TextOutA (PreCall)
     [KT:15.600100ms / UT:0.000000ms / CC:42258224]
21442852:   Parameters:
              HDC hdc [0x002DFA60] "1795229328" (unsigned dword)
              long x [0x002DFA64] "0" (signed dword)
              long y [0x002DFA68] "0" (signed dword)
              LPCSTR lpString [0x002DFA6C] "#" (ansi-string)
              long c [0x002DFA70] "19" (signed dword)
21442852:   Custom parameters:
21442852:   Stack trace:
21442852:     1) TestApplication.exe + 0x00014A91
21442852:     2) TestApplication.exe + 0x0001537E
21442852:     3) TestApplication.exe + 0x000151E0
21442852:     4) TestApplication.exe + 0x0001507D
```

Как вы видите, CTest извлекает полную информацию о типах и значениях параметров перехваченных функций. Также мы получили точное время перехвата и трассировку стека. Благодаря трассировке можно определить, из какого места тестового приложения был сделан каждый вызов. Эта информация окажется полезной, если вам нужно перехватывать только некоторые вызовы.

Для реализации нашего бота будет достаточно функциональности, которую предоставляет пример CTest. Возьмём его код за основу и

добавим в него алгоритм бота. Для этого необходимо выполнить следующие действия:

1. Откройте в Visual Studio файл проекта примера CTest. Его можно найти в архиве с исходным кодом Deviare по пути `Samples\C\Test\CTest.sln`
2. Откройте файл `MySpyMgr.cpp`, который содержит код обработки перехваченных функций.
3. В открытом файле найдите метод обработчика `CMySpyMgr::OnFunctionCalled`. Он вызывается перед тем, как управление будет передано в WinAPI-функцию. Метод достаточно длинный, но всё что в нём происходит — это вывод в лог-файл трассировки стека, параметров и возвращаемого значения перехваченной функции.
4. Перед методом `CMySpyMgr::OnFunctionCalled` добавьте функцию `ProcessParam` из листинга 5-12, реализующую алгоритм бота.

Листинг 5-12. Алгоритм бота в функции `ProcessParam`

```
VOID ProcessParam(__in Deviare2::INktParam *lpParam)
{
    CComBSTR cBstrName;
    lpParam->get_Name(&cBstrName);

    unsigned long val = 0;
    HRESULT hRes = lpParam->get_ULongVal(&val);
    if (FAILED(hRes))
        return;

    wprintf(L"ProcessParam() - name = %s value = %u\n",
            (BSTR)cBstrName, (unsigned int)(val));

    if (val < 10)
    {
        INPUT Input = { 0 };
        Input.type = INPUT_KEYBOARD;
        Input.ki.wVk = '1';
        SendInput( 1, &Input, sizeof( INPUT ) );
    }
}
```

5. В метод `CMySpyMgr::OnFunctionCalled` добавьте вызов функции `ProcessParam`. Найдите следующую строчку:

```
    if (sCmdLineParams.bAsyncCallbacks == FALSE &&
        SUCCEEDED(callInfo->Params(&cParameters)))
    {
        LogPrint(L"  Parameters:\n");
```

Замените её на это:

```
1 if (sCmdLineParams.bAsyncCallbacks == FALSE &&
2     SUCCEEDED(callInfo->Params(&cParameters)))
3 {
4     if (SUCCEEDED(cParameters->GetAt(4, &cParam)))
5         ProcessParam(cParam);
6     LogPrint(L"  Parameters:\n");
```

Разберём подробнее код вызова функции `ProcessParam`. В первом операторе `if` проверяются два условия:

1. Был ли указан ключ командной строки `-async` при запуске CTest. Если был, то параметры перехваченного вызова будут обрабатываться **асинхронно**.
2. Из объекта `callInfo` успешно удалось извлечь параметры перехваченной функции и записать их в массив объектов `cParameters`.

Если одна из этих проверок не прошла, алгоритм бота не будет вызван.

Во втором операторе `if` проверяется, что пятый параметр перехваченной функции удалось прочитать без ошибки. Он соответствует длине печатаемой строки. Этот параметр передаётся в следующий далее вызов `ProcessParam`.

В C++ все массивы нумеруются с нуля. Поэтому пятый параметр функции `TextOutA` имеет индекс четыре.

Рассмотрим алгоритм функции `ProcessParam` из листинга 5-12:

1. В переменную `cBstrName` прочитать имя пятого параметра функции `TextOutA`. Для этого используется метод `get_Name` объекта `lpParam`, в котором хранится вся информация о параметре.
2. В переменную `val` прочитать значение параметра с помощью метода `get_ULongVal` объекта `lpParam`. Если это не удалось, функция `ProcessParam` завершит свою работу.
3. Вывести в консоль имя `cBstrName` и значение `val` параметра с помощью функции `wprintf`. Этот диагностический вывод позволит проверить входные данные для следующего далее алгоритма бота.

4. Проверить, что текущее значение `val` параметра меньше десяти. Если это так, симулировать нажатие клавиши “1”.

Чтобы запустить CTest и тестовое приложение, выполните следующие шаги:

1. Скомпилируйте проект CTest под 32-разрядную платформу.
2. Получившийся бинарный файл `CTest.exe` скопируйте с заменой в каталог `deviare-bin`.
3. Скопируйте исполняемый файл тестового приложения `TestApplication.exe` в каталог `deviare-bin`.
4. Запустите приложение CTest следующей командой:

```
1 CTest.exe exec TestApplication.exe -log=out.txt
```

Иллюстрация 5-5 демонстрирует окна запущенных приложений CTest и TestApplication.

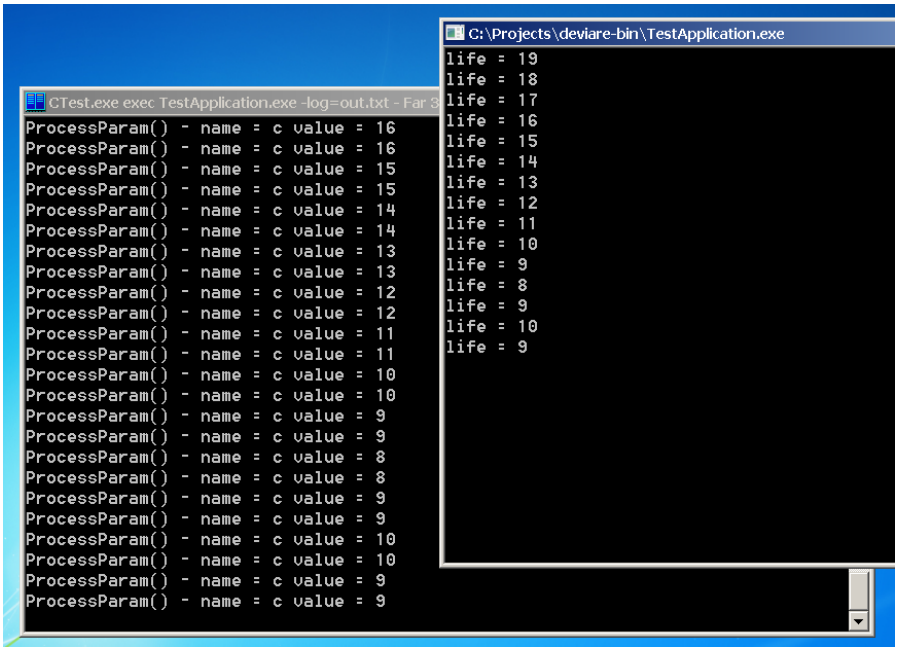


Иллюстрация 5-5. Окна приложений CTest и TestApplication

В окне TestApplication выводится текущее значение переменной `gLife`. Его же мы видим в окне CTest, но полученное из перехваченного вызова `TextOutA`. Если `gLife` опустится ниже десяти, бот будет симулировать нажатие клавиши “1”.

Выводы

Мы рассмотрели две техники перехвата данных на уровне ОС. Они позволяют получить точную информацию о состоянии игровых объектов. В то же время эти техники имеют несколько преимуществ над чтением данных из памяти процесса игры:

1. Большинство антиотладочных приёмов не защищают от перехвата WinAPI-вызовов.
2. Намного проще реализовать обработчик перехваченной функции, чем анализировать память игрового приложения.
3. Системам защиты крайне сложно обнаружить факт перехвата вызовов.

Вы можете использовать техники перехвата WinAPI-вызовов не только в алгоритме бота, но и для исследования памяти процесса игрового приложения. Они помогут вам проверить предположения об алгоритмах игры и организации её данных.

В статье рассмотрены техники перехвата WinAPI-вызовов, не упомянутые в этой книге.

Заключение

Только что мы рассмотрели последний пример. На этом заканчивается наше знакомство с ботами для компьютерных игр. Надеюсь, что вы узнали для себя что-то новое из этой книги и приятно провели с ней время. Если она вам понравилась, поделитесь ею с вашими друзьями. Также я буду вам благодарен, если вы уделите несколько минут своего времени для написания отзыва о книге на Goodreads.

Если у вас остались какие-то вопросы или возникли замечания по материалу книги, обязательно напишите о них мне на почту petrsum@gmail.com. Также вы можете задать свои вопросы в “Issues” одного из следующих GitHub репозиториев:

- https://github.com/ellysh/video-game-bots-ru
- https://github.com/Apress/practical-video-game-bots
- https://github.com/ellysh/video-game-bots

Большое вам спасибо за то, что вы прочитали “Боты для компьютерных игр”.